Nominal Logic Programming

JAMES CHENEY
University of Edinburgh
and
CHRISTIAN URBAN
Technische Universität, München

Nominal logic is an extension of first-order logic which provides a simple foundation for formalizing and reasoning about abstract syntax modulo consistent renaming of bound names (that is, α-equivalence). This article investigates logic programming based on nominal logic. This technique is especially well-suited for prototyping type systems, proof theories, operational semantics rules, and other formal systems in which bound names are present. In many cases, nominal logic programs are essentially literal translations of "paper" specifications. As such, nominal logic programming provides an executable specification language for prototyping, communicating, and experimenting with formal systems.

We describe some typical nominal logic programs, and develop the model-theoretic, proof-theoretic, and operational semantics of such programs. Besides being of interest for ensuring the correct behavior of implementations, these results provide a rigorous foundation for techniques for analysis and reasoning about nominal logic programs, as we illustrate via two examples.

Categories and Subject Descriptors: D.1.6 [**PROGRAMMING TECHNIQUES**]: Logic Programming; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs—*logics of programs, specification techniques*; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*model theory, proof theory, logic and constraint programming*

General Terms: Languages

Additional Key Words and Phrases: nominal logic, logic programming, specification, implementation

The first author was supported by AFOSR grant F49620-01-1-0298 (Next Generation Systems Languages), ONR grant N00014-01-1-0968 (Language-Based Security for Malicious Mobile Code), AFOSR grant F49620-03-1-0156 (Trust in Security-Policy Enforcement Mechanisms) and EPSRC grant R37476 while performing this research. The second author was supported by a fellowship from the Alexander-von-Humboldt foundation and an Emmy-Noether fellowship from the German Research Council.

This paper expands and improves upon material presented in several earlier publications, primarily [Gabbay and Cheney 2004; Cheney and Urban 2004; Urban and Cheney 2005; Cheney 2006b].

> Declarative Programming enables one to concentrate on the essentials of a problem, without getting bogged down in too much operational detail.
>
> David Warren in [Sterling and Shapiro 1994]

1. INTRODUCTION

As stated by Warren the ideal of logic programming is that all the programmer needs to do is describe the problem suitably, and let the computer deal with the search for solutions. Thus, logic programming languages such as Prolog are very well-suited to problem solving situations in which a problem can be formulated as a set of inference rules describing a solution. All the programmer has to do is describe the problem and ask the system to search for solutions.

Unfortunately, for some problems this ideal is not achievable in Prolog, the most well-known logic programming language, even in areas where this language is regarded as superior. Consider for example the usual three inference rules by which the type-system for lambda-terms is specified:

$$\frac{x : T \in \Gamma}{\Gamma \rhd x : T} \qquad \frac{\Gamma \rhd M : T \to T' \quad \Gamma \rhd N : T}{\Gamma \rhd MN : T'} \qquad \frac{\{x : T\} \cup \Gamma \rhd M : T'}{\Gamma \rhd \lambda x.M : T \to T'}$$

In the third rule it is often implicitly assumed that x is a variable not already present in Γ. Inferring a type for the term M has in the context Γ should fit Prolog's declarative programming paradigm very well. However, a direct, naïve implementation of such typing rules, as for example given in [Mitchell 2003, Page 489]:

$$\begin{array}{ll}
mem(X, [X|T]). & \\
mem(X, [Y|T]) & :- \ mem(X, T). \\
 & \\
type(G, var(X), T) & :- \ mem((X, T), G). \\
type(G, app(M, N), T) & :- \ type(G, M, arr(S, T)), type(G, N, S). \\
type(G, lam(X, M), arr(S, T)) & :- \ type([(X, S)|G], M, T).
\end{array}$$

behaves incorrectly on terms in which a lambda-bound name "shadows" another binding occurrence of a name. For example, typechecking the lambda-term $\lambda x.\lambda x.(x\ x)$ via the query

$$?\text{–}\ type([], lam(x, lam(x, app(var(x), var(x)))), U)$$

yields two answers, both incorrect: $U = T \to (T \to S) \to S$ and $U = (T \to S) \to T \to S$. This problem can be fixed by judicious use of the "cut" pruning operator, or by defining a `gensym` predicate, defining capture-avoiding substitution, and performing explicit α-renaming (see [Clocksin and Mellish 2003]), but both solutions rely on nonlogical, nondeclarative features of Prolog. Thus, one loses declarativeness and becomes "bogged down in operational detail" almost immediately even for the simplest problems involving name-binding.

The problems with the naïve implementation stem from the lack of support for names, name-binding and alpha-equivalence in Prolog. A number of techniques for incorporating such support into logic programming languages have been investigated, including higher-order logic programming [Nadathur and Miller 1998], Qu-

Prolog [Staples et al. 1996], and logic programming with binding algebras [Hamana 2001].

Of these approaches, higher-order logic programming may be the most convenient and compelling. For example, the typechecking relation can be implemented in λProlog as follows:

$$\begin{array}{ll} tc\ (app\ E_1\ E_2)\ U & :-\ tc\ E_1\ (arr\ T\ U), tc\ E_2\ T. \\ tc\ (lam\ (\lambda x.E\ x))\ (arr\ T\ U) & :-\ \Pi x.\ tc\ x\ T \Rightarrow tc\ (E\ x) U. \end{array}$$

Here, meta-language variables and λ-bindings are used to represent object-language variables and bindings; object language application and lambda-abstraction are represented using constants $app : exp \to exp \to exp$ and $lam : (exp \to exp) \to exp$. Moreover, local parameters (introduced using the universal quantifier, written Π) and local assumptions (introduced using the implication subgoal ⇒) are used to represent the scope restrictions on the local variable and its type assumption. Thus, the meta-language's context is used to implement locally-scoped parameters and hypotheses of the object language.

This is a very elegant technique for programming and reasoning about programming languages. Unfortunately, there are some situations in which higher-order encodings are no simpler than first-order equivalents; sometimes, the use of higher-order features even obstructs natural-seeming programming techniques. As a case in point, consider a simplistic *closure conversion* translation, which eliminates local parameters from lambda-calculus expressions:

$$\begin{array}{lll} C[\![x, \Gamma \vdash x]\!]e & = & \pi_1(e) \\ C[\![y, \Gamma \vdash x]\!]e & = & C[\![\Gamma \vdash x]\!](\pi_2(e)) \qquad (x \neq y) \\ C[\![\Gamma \vdash t_1 t_2]\!]e & = & \mathsf{let}\ c\ =\ C[\![\Gamma \vdash t_1]\!]e\ \mathsf{in}\ (\pi_1(c))\ (C[\![\Gamma \vdash t_2]\!]e, \pi_2(c)) \qquad (c \notin FV(e)) \\ C[\![\Gamma \vdash \lambda x.t]\!]e & = & (\lambda y. C[\![x, \Gamma \vdash t]\!]y, e) \qquad (x, y \notin \Gamma) \end{array}$$

One can clearly implement this translation using explicit name-generation and substitution in Prolog; since names are just ground atoms, the inequality test in the second defining equation is definable as Prolog's built-in inequality predicate. But if we use higher-order abstract syntax, there is no need to perform explicit renaming or name-generation, but the fact that object-language variables "disappear" makes it difficult to provide correct behavior. For example, the following higher-order logic program solves this problem for the $lam : (exp \to exp) \to exp$ representation of the lambda calculus. It is the simplest solution we have been able to develop.

$$\begin{array}{l} cconv\ (X :: G)\ X\ Env\ (pi1\ Env) :- isVar\ X. \\ cconv\ (Y :: G)\ X\ Env\ T :- member\ X\ G, cconv\ G\ X\ (pi2\ Env)\ T. \\ cconv\ G\ (app\ T\ U)\ Env\ (let\ T'\ (\lambda c.app\ (pi1\ c)\ (pair\ U'\ (pi2(c))))) \\ \quad :- cconv\ G\ T\ Env\ T', cconv\ G\ U\ Env\ U'. \\ cconv\ G\ (lam(\lambda x.T\ x))\ Env\ (pair\ (lam(\lambda y.T\ y))\ Env) \\ \quad :- \Pi x.\ isVar\ x \Rightarrow \Pi y.\ cconv\ (x :: G)\ (T\ x)\ y\ (T'\ y) \end{array}$$

In order to be able to distinguish variables of type *exp* from other terms, we need to add local hypotheses $isVar\ x$ whenever we traverse a λ. In the second clause, we exploit the fact that for well-formed terms, X and Y are distinct variables and Y appears first in the context if and only if X appears later in the context.

While higher-order techniques have many advantages for implementing formal

systems involving binding and substitution, they do not appear well suited to situations where "low-level" access to names as first-class data is required. While it is clearly a matter of taste whether the higher-order implementation of *cconv* is tolerable, it is indisputable that the higher-order implementation of *cconv* departs significantly from what one would write on paper. Instead it is sometimes necessary to translate between the informal syntactic definitions people seem to find intuitive and the formal notations which the language provides. Because of this *impedance mismatch*, in neither first-order nor higher-order logic programming is it always possible to simply "concentrate on the essentials of a problem" involving names and binding.

In this paper, we investigate a new approach in which both of the above examples (and a wide variety of other programs) can be implemented easily and (we argue) intuitively. Our approach is based on *nominal logic*, an extension of first-order logic introduced by Pitts [2003], and based on the novel approach to abstract syntax developed by Gabbay and Pitts [2002]. In essence, nominal logic axiomatizes an inexhaustible collection of *names* x, y and provides a first-order axiomatization of a name-binding operation $\langle \mathsf{x} \rangle t$ (called *abstraction*) in terms of two primitive operations, *swapping* $((a\ b) \cdot t)$ and *freshness* $(a \# t)$. In addition, nominal logic includes a novel quantified formula $И\mathsf{a}.\phi$ ("for fresh a, ϕ holds") which quantifies over fresh names.

In nominal logic, names and binding are abstract data types admitting only swapping, binding, and operations for equality and freshness testing. Name-abstractions $\langle \mathsf{x} \rangle t$ are considered equal up to α-equivalence, defined in terms of swapping and freshness. For example, object variables x and lambdas $\lambda x.t$ can be encoded as nominal terms $var(\mathsf{x})$ and abstractions $lam(\langle \mathsf{x} \rangle t)$ where $var : id \to exp$ and $lam : \langle id \rangle exp \to exp$. We can obtain a correct implementation of the *type* relation above by replacing the third clause with

$$type(G, lam(\langle \mathsf{x} \rangle E), arrTy(T, U)) :- x \# G, type((x, T) :: G, E, U).$$

which we observe corresponds closely to the third inference rule (reading $lam(\langle \mathsf{x} \rangle E)$ as $\lambda x.E$, $x \notin FV(\Gamma)$ as $x \# G$, and $\{x{:}\tau\} \cup \Gamma$ as $(x, T) :: G$). We revisit this example and the closure conversion function in Section 2.

We refer to this approach to programming with names and binding modulo α-equivalence as *nominal abstract syntax*. This approach provides built-in α-equivalence and fresh name generation, while retaining a clear declarative interpretation. Names are sufficiently abstract that the low-level details of name generation can be hidden from the programmer, yet still sufficiently concrete that there is no difficulty working with open terms, freshness constraints, or inequalities among names precisely as is done "on paper". Nominal abstract syntax and nominal logic make possible a distinctive new style of meta-programming, which we call *nominal logic programming*.

Unlike some previous approaches (particularly higher-order logic programming and Qu-Prolog), capture-avoiding substitution is not "built-in"; however, this is not as serious a problem as it might seem, since substitution can be defined declaratively in nominal logic. There is no obstacle to adding a built-in capture-avoiding substitution operator, but it is not needed for defining α-equivalence, so there is also no reason why this complex operation has to be built into the language defi-

nition. We feel this is an advantage, not a disadvantage; moreover, the additional effort that needs to be expended on implementing substitution could be avoided using other techniques such as generic programming [Cheney 2005c] which are of independent interest.

In this paper, we describe a particular implementation of nominal logic programming, called αProlog. We also investigate the semantics of nominal logic programs and discuss applications of these results.

—We first (Section 2) illustrate nominal logic programming via several examples written in αProlog, drawing on familiar examples based on the λ-calculus and π-calculus. The aim of these examples is to show that, in contrast to all other known approaches, αProlog programs can be used to encode calculi correctly yet without essential alterations to their paper representations. Thus, αProlog can be used as a lightweight prototyping tool by researchers developing new systems, or by students learning about existing systems.

—We next (Section 3) provide a summary of nominal abstract syntax and nominal logic needed for the rest of the paper. We introduce the domain of nominal terms, which plays a similar role to ordinary first-order terms in Prolog or lambda-terms in λProlog, and then review the semantics of term models of nominal logic (previously developed in [Cheney 2006a]).

—Section 4 develops the semantics of nominal logic programs. This is crucial for justifying our claim that the notation and concepts of nominal logic match our intuition, and that nominal logic programs capture the informal meaning we assign to them. Using the foundations introduced in Section 3, we provide a model-theoretic semantics of nominal logic programs following Lloyd [1987]. We also introduce a proof-theoretic semantics via a variation of the proof-theoretic semantics of CLP, investigated by Darlington and Guo [1994] and Leach et al. [2001]. Finally, we present an operational semantics that models the low-level proof search behavior of an interpreter more directly. We prove appropriate soundness and completeness results relating these definitions along the way.

—In Section 5, we consider some applications of the semantics to issues arising in an implementation such as αProlog. We verify the correctness of a standard "elaboration" transformation and an optimization which permits us to avoid having to solve expensive, NP-complete nominal constraint solving problems during execution. This result supersedes a similar result of Urban and Cheney [2005].

—Section 6 presents a detailed comparison of our work with previous techniques for incorporating support for name-binding into programming languages and Section 7 concludes.

In order to streamline the exposition, many routine cases in proofs in the body of the paper have been omitted. Complete proofs are available in appendices.

2. PROGRAMMING IN αPROLOG

2.1 Syntax

Before presenting examples, we sketch the syntax which is used for declarations of constants, function symbols, types and type abbreviations, clause declarations, and queries in this paper. We also mention some other standard conveniences provided

Terms	t, u	$::=$	$X \mid c \mid f(\vec{t}) \mid \mathsf{a} \mid \langle a \rangle t \mid (a\ b) \cdot t \mid i \mid$ 'c' $\mid [] \mid t :: t' \mid [t_1, \ldots, t_n \mid t'] \mid (t, t')$
Constructor types	τ, ν	$::=$	$tid\ \vec{\sigma} \mid \sigma \to \tau$
Types	σ	$::=$	$\alpha \mid \tau \mid \langle \nu \rangle \sigma \mid \mathbf{int} \mid \mathbf{char} \mid \mathbf{list}\ \sigma \mid \sigma \times \sigma' \mid \sigma \to \sigma \mid o$
Basic Kinds	κ_0	$::=$	$\mathbf{name\_type} \mid \mathbf{type}$
Kinds	κ	$::=$	$\kappa_0 \mid \kappa_0 \to \kappa$
Atomic formulas	A	$::=$	$p(\vec{t}) \mid f(\vec{t}) = u$
Goals	G	$::=$	$A \mid a \# t \mid t \approx u \mid G, G' \mid G; G' \mid \exists X{:}\sigma.G \mid \nu \mathsf{a}{:}\nu.G$
Declarations	D	$::=$	$tid : \kappa \mid defid :: \sigma \mid \mathbf{type}\ tid\ \vec{\alpha} = \sigma \mid conid : \tau \mid A :\!\!- G$

Fig. 1. Syntax summary

by the αProlog implementation such as polymorphism, definite clause grammars and defined function symbols.

The syntax we shall employ is outlined in Figure 1. To improve readability, the syntax employed in the paper differs from the ASCII syntax employed in the current implementation. The nominal terms used in αProlog include standard first-order variables X, constants c, and function symbols f; also, we have new syntax for names a, name-abstractions $\langle a \rangle t$, and swappings $(a\ b) \cdot t$.

Names and name-abstractions are used to represent syntax with bound names in αProlog. The unification algorithm used by αProlog solves equations modulo an equational theory that equates terms modulo α-renaming of names bound using abstraction. Swappings are a technical device (similar to explicit substitutions [Abadi et al. 1991; Dowek et al. 1998]) which are needed in constraint solving. We will present the details of the equational theory in Section 3.

αProlog also contains standard built-in types for pairing, lists, integers, and characters. Note that $[t_1, \ldots, t_n \mid t']$ is a standard Prolog notation for matching against an initial segment of a list; it is equivalent to $t_1 :: \cdots :: t_n :: t'$.

User-defined types, including name types, can be introduced using declarations such as

$$tid : \mathbf{type}. \qquad ntid : \mathbf{name\_type}.$$

Also, using functional kinds, we can introduce new type constructors used for user-defined parameterized types. For example, **list** could be declared as

$$\mathbf{list} : \mathbf{type} \to \mathbf{type}.$$

Similarly, abstraction $\langle \nu \rangle \sigma$ could be declared as

$$\langle - \rangle - : \mathbf{name\_type} \to \mathbf{type} \to \mathbf{type}.$$

Only first-order kinds are supported in the current implementation.

Type abbreviations (possibly with parameters) can be introduced using the syntax

$$\mathbf{type}\ tid\ \alpha_1\ \cdots\ \alpha_n = \sigma(\alpha_1, \ldots, \alpha_n).$$

Likewise, uninterpreted constants and function symbols (which we call *(term) constructors*) are declared using a similar notation:

$$conid : \tau.$$

here τ is a "constructor type", that is, either a user-defined type constructor application $tid\ \vec{\sigma}$ or a function type returning a constructor type. These restrictions

ensure that user-defined term constructors cannot be added to built-in types, including name-types, lists and products. Constants and function symbols must return a user-defined data type; so, there can be no constants, function symbols, or other user-defined terms in a name type, only name-constants.

Interpreted function and predicate symbols can be defined using the syntax

$$defid :: \sigma$$

for example,

$$p :: \sigma \times \sigma \to o \qquad f :: \sigma \to \sigma$$

introduce constants for a binary relation p on type σ and a unary function f on type σ. There is no restriction on the return types of defined symbols.

As in Prolog, programs are defined using Horn clauses $A :- G$ where A is an atomic formula and G is a goal formula. Atomic formulas include user-defined predicates $p(\vec{t})$ as well as equations $f(\vec{t}) = u$; in either case p or f must be a defined symbol of appropriate type, not a constructor.

Goal formulas G can be built up out of atomic formulas A, freshness constraints $a \# t$, equations $t \approx u$, conjunctions G, G', disjunctions $G; G'$, existential quantification $\exists X.G$, or И-quantification $И\mathsf{a}.G$. The freshness constraint $a \# t$ holds if the name a does not appear free (that is, outside an abstraction) in t; equality $t \approx u$ between nominal terms is modulo α-renaming of name-abstractions. For example, $\langle \mathsf{a} \rangle (\mathsf{a}, \mathsf{b}) \approx \langle \mathsf{c} \rangle (\mathsf{c}, \mathsf{b}) \not\approx \langle \mathsf{b} \rangle (\mathsf{b}, \mathsf{b})$.

As usual in logic programming, we interpret a program clause $A :- G$ with free variables $\vec{X}$ as an implicitly quantified, closed formula $\forall \vec{X}.G \Rightarrow A$. Moreover, if the program clause contains free names $\vec{\mathsf{a}}$, they are interpreted as implicitly И-quantified outside of the scope of the universally-quantified variables:

$$И\vec{\mathsf{a}}.\forall \vec{X}.G \Rightarrow A$$

Standard extensions. The current implementation of αProlog provides a goal **not**(G) that searches for a derivation of G, fails if one is found and succeeds otherwise; a "cut" goal ! that prunes alternative choice points for the current subgoal, and "guard" goal $G \to G_1 | G_2$ which attempts to solve G, proceeds to G_1 if successful, and proceeds to G_2 if not. These are standard, though each can damage declarative transparency and they are not considered in the semantics. αProlog also provides support for *definite clause grammars*, a simple, yet powerful parsing mechanism; as in Prolog, definite clause grammars are provided as a source to source translation.

Polymorphism. αProlog permits type variables in declarations, which are treated polymorphically, following previous work on polymorphic typing in logic programming [Mycroft and O'Keefe 1984; Hanus 1991]. Polymorphic type checking is performed in the standard way by generating equational constraints and solving them using unification. As observed by Hanus, handling general polymorphism in logic programming may require performing typechecking at run-time. To avoid this, the current implementation αProlog rules out "non-parametric" polymorphic

program clauses that specialize type variables. For example, the second clause in

$$\begin{array}{ll} \mathit{head} & :: \alpha \times \mathbf{list}\ \alpha \to o. \\ \mathit{head}(X, X :: L). & \\ \mathit{head}(1, 1 :: L). & \end{array}$$

works only for $\alpha = \mathit{int}$, not for arbitrary α, so is ruled out.

A simple subkinding system is used to assign either kind **type** or **name_type** to type variables. In combination with the subkinding needed to relate name-types and ordinary types; however, we consider only the semantics of monomorphic programs.

Function definitions. As in other Prolog-like languages, it is often convenient to have a notation for writing predicates which are easier written as functions. For example, the functional definition

$$\begin{array}{lll} \mathit{append} & :: & \mathbf{list}\ \alpha \times \mathbf{list}\ \alpha \to \mathbf{list}\ \alpha. \\ \mathit{append}([], M) & = & M. \\ \mathit{append}(X :: L, M) & = & X :: \mathit{append}(L, M). \end{array}$$

can be viewed as an abbreviation for the relational definition

$$\begin{array}{lll} \mathit{appendp} & :: & \mathbf{list}\ \alpha \times \mathbf{list}\ \alpha \times \mathbf{list}\ \alpha \to o. \\ \mathit{appendp}([], M, M). & & \\ \mathit{appendp}(X :: L, M, X :: N) & :- & \mathit{appendp}(L, M, N). \end{array}$$

Using notation for functional definitions can considerably simplify a program.

It is a standard exercise in logic to translate a theory with both defined functions and relation symbols to a logically equivalent, purely relational theory; essentially, wherever we encounter an occurrence of the defined function, we replace it with a variable that is appropriately constrained by a corresponding relation. Concretely, this means that a goal such as

$$\mathit{append}(\mathit{append}([1], X), [2]) \approx Y$$

is translated first to

$$\exists Z.\mathit{appendp}([1], X, Z), \mathit{append}(Z, [2]) \approx Y$$

and then to

$$\exists Z.\mathit{appendp}([1], X, Z), \exists W.\mathit{appendp}(Z, [2], W), W \approx Y$$

This technique for implementing functions in logic programming is called *flattening* [Hanus 1994]. More sophisticated techniques such as *narrowing* that have been investigated in functional logic programming could also be used; however, doing so will require extending equational unification techniques to nominal logic.

2.2 The λ-calculus and variants

The prototypical example of a language with variable binding is the λ-calculus. In αProlog, the syntax of λ-terms may be described with the following type and constructor declarations:

$$\begin{array}{lll} \mathit{id} : \mathbf{name\_type}. & \mathit{exp} : \mathbf{type}. & \\ \mathit{var} : \mathit{id} \to \mathit{exp}. & \mathit{app} : \mathit{exp} \times \mathit{exp} \to \mathit{exp}. & \mathit{lam} : \langle \mathit{id} \rangle \mathit{exp} \to \mathit{exp}. \end{array}$$

$$\begin{array}{lll} \text{Terms} & e & ::= x \mid \lambda x.e \mid e\ e' \\ \text{Types} & \tau & ::= b \mid \tau \to \tau' \\ \text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, x{:}\tau \end{array} \qquad \begin{array}{rcll} x\{e/x\} & = & e & \\ y\{e/x\} & = & y & (x \neq y) \\ (e_1\ e_2)\{e/x\} & = & e_1\{e/x\}\ e_2\{e/x\} & \\ (\lambda y.e')\{e/x\} & = & \lambda y.e'\{e/x\} & (y \notin FV(x,e)) \end{array}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau' \quad (x \notin Dom(\Gamma))}{\Gamma \vdash \lambda x.t : \tau \to \tau'}$$

Fig. 2. Lambda-calculus: syntax, substitution, and typing

We make the simplifying assumption that the variables of object λ-terms are constants of type *id*. Then we can translate λ-terms as follows:

$$\ulcorner \mathsf{x} \urcorner = var(\mathsf{x}) \qquad \ulcorner e_1\ e_2 \urcorner = app(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \qquad \ulcorner \lambda \mathsf{x}.e \urcorner = lam(\langle \mathsf{x} \rangle \ulcorner e \urcorner)$$

It is not difficult to verify (using the semantic techniques we present in Section 4) that the encoding function $\ulcorner \cdot \urcorner$ is a bijection between (open) λ-terms modulo α-equivalence and nominal terms of type *exp*, i.e. $e \equiv_\alpha e'$ if and only if $\ulcorner e \urcorner \approx \ulcorner e' \urcorner$. Similarly,

$$\begin{array}{rcl} \mathsf{x} \notin FV(e) & \iff & \mathsf{x} \# \ulcorner e \urcorner \\ \ulcorner e\{\mathsf{x}/\mathsf{y}, \mathsf{y}/\mathsf{x}\} \urcorner & = & (\mathsf{x}\ \mathsf{y}) \cdot \ulcorner e \urcorner \end{array}$$

These properties help ensure the *adequacy* of this nominal abstract syntax encoding.

Example 2.1 *Typechecking and inference.* First, for comparison with higher-order encodings, we consider the problem of typechecking λ-terms. The syntax of types can be encoded as follows:

$$tid : \textbf{name\_type}. \qquad ty : \textbf{type}. \qquad varTy : tid \to ty. \qquad arrTy : ty \times ty \to ty.$$

We define contexts *ctx* as lists of pairs of identifiers and types, and the 3-ary relation *typ* relating a context, term, and type:

$$\begin{array}{lcl} \textbf{type}\ ctx & = & \textbf{list}\ (id \times ty). \\ tc & :: & ctx \times exp \times ty \to o. \\ tc(C, var(X), T) & :- & mem((X,T), C). \\ tc(C, app(E_1, E_2), T') & :- & tc(C, E_1, arrTy(T, T')), tc(C, E_2, T). \\ tc(C, lam(\langle \mathsf{x} \rangle E), arrTy(T, T')) & :- & \mathsf{x} \# C, tc([(\mathsf{x}, T)|C], E, T'). \end{array}$$

The predicate $mem :: \alpha \times [\alpha] \to o$ is the usual predicate for testing list membership ($x : \tau \in \Gamma$). The side-condition $x \notin Dom(\Gamma)$ is translated to the freshness constraint $\mathsf{x} \# C$. Given that the constraint $x \notin Dom(\Gamma)$ is often left implicit in informal presentations of the inference rules for λ-term typing, the αProlog version of these rules is about as close to a "paper" presentation as one can get.

Consider the query $?\text{–}\ tc([], lam(\langle \mathsf{x} \rangle lam(\langle \mathsf{y} \rangle var(\mathsf{x}))), T)$. We can reduce this goal by backchaining against the suitably freshened rule

$$tc(C_1, lam(\langle \mathsf{x}_1 \rangle E_1), arr(T_1, U_1)) :- \mathsf{x}_1 \# C_1, tc([(\mathsf{x}_1, T_1)|C_1], E_1, U_1)$$

which unifies with the goal with $[C_1 = [], E_1 = lam(\langle \mathsf{y} \rangle var(\mathsf{x}_1)), T = arr(T_1, U_1)]$. This yields subgoal $\mathsf{x}_1 \# [], tc([(\mathsf{x}_1, T_1)|C_1], E_1, U_1)$. The first conjunct is trivially valid since C_1 is a constant. The second is solved by backchaining against the third

typ-rule again, producing unifier $[C_2 = [(\mathsf{x}_1, T_1)], E_2 = var(\mathsf{x}_1), U_1 = arr(T_2, U_2)]$ and subgoal $\mathsf{x}_2 \# [(\mathsf{x}_1, T_1)], tc([(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], var(\mathsf{x}_1), U_2)$. The freshness subgoal reduces to the constraint $\mathsf{x}_2 \# T_1$, and the *typ* subgoal can be solved by backchaining against

$$tc(C_3, var(X_3), T_3) :- mem((X_3, T_3), C_3)$$

using unifier $[C_3 = [(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], X_3 = \mathsf{x}_1, T_3 = U_2]$. Finally, the remaining subgoal $mem((\mathsf{x}_1, U_2), [(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)])$ clearly has most general solution $[U_2 = T_1]$. Solving for T, we have $T = arr(T_1, U_1) = arr(T_1, arr(T_2, U_2)) = arr(T_1, arr(T_2, T_1))$. This solution corresponds to the principal type of $\lambda x.\lambda y.x$.

There are no other possible solutions.

Example 2.2 *Capture-avoiding substitution.* Although capture-avoiding substitution is not a built-in operator in αProlog, it is easy to define via the clauses:

$$\begin{array}{lcl}
subst & :: & exp \times exp \times id \to exp. \\
subst(var(X), E, X) & = & E. \\
subst(var(Y), E, X) & = & var(Y) \\
 & :- & X \# Y. \\
subst(app(E_1, E_2), E, X) & = & app(subst(E_1, E, X), subst(E_2, E, X)). \\
subst(lam(\langle \mathsf{y} \rangle E'), E, X) & = & lam(\langle \mathsf{y} \rangle subst(E', E, X)) \\
 & :- & \mathsf{y} \# (X, E).
\end{array}$$

Note the two freshness side-conditions: the constraint $X \# Y$ prevents the first and second clauses from overlapping; the constraint $\mathsf{y} \# (X, E)$ ensures capture-avoidance, by restricting the application of the fourth clause to when y is fresh for X and E. Despite these side-conditions, this definition is total and deterministic. Determinism is immediate: no two clauses overlap. Totality follows because, by nominal logic's *freshness principle*, the bound name y in $lam(\langle \mathsf{y} \rangle E')$ can always be renamed to a fresh z chosen so that $\mathsf{z} \# (X, E)$. It is straightforward to prove (using the semantics we will provide in Section 4) that $subst(\ulcorner t \urcorner, \ulcorner t' \urcorner, \mathsf{x})$ coincides with the traditional capture-avoiding substitution on λ-terms $t[t'/x]$; that is, $\ulcorner t[t'/\mathsf{x}] \urcorner = subst(\ulcorner t \urcorner, \ulcorner t' \urcorner, \mathsf{x})$.

Consider the goal ?– $X = subst(lam(\langle \mathsf{x} \rangle var(\mathsf{y})), var(\mathsf{x}), \mathsf{y})$. The substitution on the right-hand side is in danger of capturing the free variable $var(\mathsf{x})$. How is capture avoided in αProlog? First, recall that function definitions are translated to a flattened clausal form in αProlog, so we must solve the equivalent goal

$$substp(lam(\langle \mathsf{x} \rangle var(\mathsf{y})), var(\mathsf{x}), \mathsf{y}, X)$$

subject to an appropriately translated definition of *substp*. The freshened, flattened clause

$$substp(lam(\langle \mathsf{y}_1 \rangle E_1'), E_1, X_1, lam(\mathsf{y}_1, E_1'')) :- \mathsf{y}_1 \# E_1, substp(E_1', E_1, X_1, E_1'')$$

unifies with substitution

$$[E_1' = var(\mathsf{y}), X_1 = \mathsf{y}, E_1 = var(\mathsf{x}), X = lam(\langle \mathsf{y}_1 \rangle E_1'')].$$

The freshness constraint $\mathsf{y}_1 \# var(\mathsf{x})$ guarantees that $var(\mathsf{x})$ cannot be captured. It is easily verified, so the goal reduces to $substp(var(\mathsf{y}), var(\mathsf{x}), \mathsf{y}, E_1'')$. Using

$$
\begin{array}{lcl}
\textbf{type}\ ctx & = & \textbf{list}\ id. \\
cconv & :: & ctx \times exp \times exp \to exp. \\
cconv([X|G], var(X), Env) & = & pi_1(Env). \\
cconv([X|G], var(Y), Env) & = & cconv(G, var(\mathsf{y}), pi_2(Env)) \\
 & :- & X \# Y. \\
cconv(G, app(E_1, E_2), Env) & = & let(cconv(G, E_1, Env), \langle \mathsf{c} \rangle \\
 & & \quad app(pi_1(var(\mathsf{c})), \\
 & & \quad\ pair(cconv(G, E_2, Env), pi_2(var(\mathsf{c}))))) \\
 & :- & \mathsf{c} \# Env. \\
cconv(G, lam(\langle \mathsf{x} \rangle E), Env) & = & pair(lam(\langle \mathsf{y} \rangle cconv([\mathsf{x}|G], E, var(\mathsf{y}))), Env) \\
 & :- & \mathsf{x} \# G, \mathsf{y} \# G.
\end{array}
$$

Fig. 3. Untyped closure conversion in αProlog

the freshened rule $substp(var(X_2), E_2, X_2, E_2)$ with unifying substitution $[X_2 = \mathsf{y}, E_2 = var(\mathsf{x}), E_1'' = var(\mathsf{x})]$, we obtain the solution $X = lam(\langle \mathsf{y}_1 \rangle var(\mathsf{x}))$.

2.2.1 *Units, pairs and closure conversion.* We can easily add syntax for unit and pair types and terms and let-bindings as follows:

$$
\begin{array}{ll}
 & unit\ :\ exp \\
unitTy\ :\ ty & pair\ :\ exp \times exp \to exp \\
pairTy\ :\ ty \times ty \to ty & pi_1\ :\ exp \to exp \\
 & pi_2\ :\ exp \to exp
\end{array}
\qquad let\ :\ exp \times \langle id \rangle exp \to exp
$$

With this addition, it is also possible to define a *closure conversion* translation from well-formed λ-terms to terms in which all subexpressions of function type are *closed* (that is, contain no references to variables declared outside the enclosing function scope). Closed functions can be lifted to the top level, so closure conversion is an important step in generating intermediate code in compilers for functional languages.

A simple closure conversion translation can be defined informally as follows:

$$
\begin{array}{lcll}
C[\![x, \Gamma \vdash x]\!]e & = & \pi_1(e) & \\
C[\![y, \Gamma \vdash x]\!]e & = & C[\![\Gamma \vdash x]\!](\pi_2(e)) & (x \neq y) \\
C[\![\Gamma \vdash t_1 t_2]\!]e & = & \mathsf{let}\ c\ =\ C[\![\Gamma \vdash t_1]\!]e\ \mathsf{in}\ (\pi_1(c))\ (C[\![\Gamma \vdash t_2]\!]e, \pi_2(c)) & (c \notin FV(e)) \\
C[\![\Gamma \vdash \lambda x.t]\!]e & = & (\lambda y. C[\![x, \Gamma \vdash t]\!]y, e) & (x, y \notin \Gamma)
\end{array}
$$

Figure 3 shows how to implement this translation in αProlog.

2.2.2 *Typed closure conversion.* The translation in Figure 3 is "untyped" in the sense that the type of the output term depends on both the input type and the current context. Minamide et al. [1996] investigated typed closure conversion algorithms that employ existential types to provide a uniform type translation. To provide a simple form of such a translation, we need to add existential types and corresponding terms to the language:

$$
\begin{array}{ll}
existsTy\ :\ \langle tid \rangle ty \to ty. & \\
 & \\
pack\ :\ ty \times exp \to exp. & \\
unpack\ :\ exp \times \langle tid \rangle \langle id \rangle exp \to exp. &
\end{array}
$$

$$
\begin{array}{lcl}
tyT & :: & ty \to ty. \\
tyT(varTy(A)) & = & varTy(A). \\
tyT(arrTy(T,U)) & = & existsTy(\langle \mathsf{a} \rangle pairTy(arrTy(pairTy(tyT(T), varTy(\mathsf{a})), tyT(U)), \\
 & & \quad varTy(\mathsf{a}))) \\
 & :- & \mathsf{a} \# (T,U).
\end{array}
$$

$$
\begin{array}{lcl}
ctxT & :: & ctx \to ty. \\
ctxT([]) & = & unitTy. \\
ctxT([(X,T)|G]) & = & pairTy(tyT(T), ctxT(G)).
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{type}\ ctx & = & \textbf{list}\ (id \times ty). \\
tcconv & :: & ctx \times exp \times ty \times exp \to exp. \\
tcconv([(X,T)|G], var(X), T, Env) & = & pi_1(Env). \\
tcconv([(X,T)|G], var(Y), U, Env) & = & tcconv(G, var(\mathsf{y}), U, pi_2(Env)) \\
 & :- & X \# Y. \\
tcconv(G, app(E_1,E_2), U, Env) & = & unpack(tcconv(G, E_1, arrTy(T,U), Env), \langle \mathsf{a} \rangle \langle \mathsf{c} \rangle \\
 & & \quad app(pi_1(var(\mathsf{c})), \\
 & & \quad\quad pair(tcconv(G, E_2, T, Env), pi_2(var(\mathsf{c}))))) \\
 & :- & \mathsf{c} \# Env. \\
tcconv(G, lam(\langle \mathsf{x} \rangle E), arrTy(T,U), Env) & = & pack(ctxT(G), \\
 & & \quad pair(lam(\langle \mathsf{y} \rangle \\
 & & \quad\quad tcconv([(\mathsf{x},T)|G], E, U, var(\mathsf{y}))), Env)) \\
 & :- & \mathsf{x} \# G, \mathsf{y} \# G.
\end{array}
$$

Fig. 4. Typed closure conversion in αProlog

We also need to redesign the *tc* judgment to include both a type and term environment. This is entirely straightforward. Once this is done, we can implement a simple typed closure conversion

$$
\begin{array}{lcll}
C[\![x{:}\tau, \Gamma \vdash x : \tau]\!]e & = & \pi_1(e) & \\
C[\![y{:}\tau', \Gamma \vdash x : \tau]\!]e & = & C[\![\Gamma \vdash x : \tau]\!](\pi_2(e)) & (x \neq y) \\
C[\![\Gamma \vdash t_1 t_2 : \tau']\!]e & = & \mathsf{unpack}\ [\alpha, c] = C[\![\Gamma \vdash t_1 : \tau \to \tau']\!]e & \\
 & & \mathsf{in}\ (\pi_1(c))\ \langle C[\![\Gamma \vdash t_2 : \tau]\!]e, \pi_2(c) \rangle & (c \notin FV(e)) \\
C[\![\Gamma \vdash \lambda x.t : \tau \to \tau']\!]e & = & \mathsf{pack}[T[\![\Gamma]\!], \langle \lambda y. C[\![x{:}\tau, \Gamma \vdash t : \tau']\!]y, e \rangle] & (x, y \notin \Gamma)
\end{array}
$$

where the type translation (used in the third case) is defined as

$$
\begin{array}{rcl}
T[\![\alpha]\!] & = & \alpha \\
T[\![\tau_1 \to \tau_2]\!] & = & \exists \alpha. (T[\![\tau_1]\!] \times \alpha \to T[\![\tau_2]\!]) \times \alpha \\
T[\![\cdot]\!] & = & unit \\
T[\![x{:}\tau, \Gamma]\!] & = & T[\![\tau]\!] \times T[\![\Gamma]\!]
\end{array}
$$

as shown in Figure 4. Again, this is a straightforward translation from the informal version.

2.2.3 *Extending to the $\lambda\mu$-calculus.* The $\lambda\mu$-calculus, introduced by Parigot [1992], extends the λ-calculus with *continuations* α; terms may be "named" by continuations ($[\alpha]e$) and continuations may be introduced with μ-binding ($\mu\alpha.e$). Intuitively, $\lambda\mu$-terms are proof terms for classical natural deduction, and μ-abstractions represent proofs by double negation. In addition to capture-avoiding substitution of terms for variables, the $\lambda\mu$-calculus introduces a capture-avoiding *replacement*

Terms, Types, and Contexts

$$
\begin{array}{lcl}
e & ::= & x \mid (e\ e') \mid \lambda x.e \mid [\alpha]e \mid \mu\alpha.e \\
\tau & ::= & b \mid \tau \to \tau' \mid \bot \\
\Gamma & ::= & \cdot \mid \Gamma, x:\tau \mid \Gamma, \alpha:\overline{\tau}
\end{array}
$$

Replacement Operation

$$
\begin{array}{rcl}
x\{e/\alpha\} & = & x \\
(e_1\ e_2)\{e/\alpha\} & = & (e_1\{e/\alpha\}\ e_2\{e/\alpha\}) \\
(\lambda y.e')\{e/\alpha\} & = & \lambda y.e'\{e/\alpha\} \\
([\alpha]e')\{e/\alpha\} & = & [\alpha](e'\{e/\alpha\}\ e) \\
([\beta]e')\{e/\alpha\} & = & [\beta](e'\{e/\alpha\}) \quad (\beta \neq \alpha) \\
(\mu\beta.e')\{e/\alpha\} & = & \mu\beta.e'\{e/\alpha\} \quad (\beta \notin FN(e,\alpha))
\end{array}
$$

Some Typing-Rules

$$\frac{\alpha{:}\tau \in \Delta \quad \Gamma \vdash e : \tau \mid \Delta}{\Gamma \vdash [\alpha]e : \bot \mid \Delta} \qquad \frac{\Gamma \vdash e_1 : \bot \mid \Delta \quad \Gamma \vdash e_2 : \tau \mid \Delta}{\Gamma \vdash (e_1\ e_2) : \bot \mid \Delta} \qquad \frac{\Gamma \vdash e : \bot \mid \Delta, \alpha{:}\tau \quad (\alpha \notin Dom(\Delta))}{\Gamma \vdash \mu\alpha.e : \tau \mid \Delta}$$

Fig. 5. A variant of Parigot's $\lambda\mu$-calculus.

operator $e'\{e/\alpha\}$ which replaces each occurrence of the pattern $[\alpha]e_0$ in e' with $[\alpha](e_0\ e)$. We give a variant of the $\lambda\mu$-calculus in Figure 5.

We may extend the λ-calculus encoding with a new name-type *con* for continuations and term constructors for $\lambda\mu$-terms:

$$con : \mathbf{name\_type} \qquad pass : (con, exp) \to exp \qquad mu : \langle con\rangle exp \to exp$$

and encoding $\ulcorner[\alpha]t\urcorner = pass(\alpha, \ulcorner t\urcorner)$ and $\ulcorner\mu\alpha.t\urcorner = mu(\langle\alpha\rangle\ulcorner t\urcorner)$. Again, it is easy to show that ground *exp*-terms are in bijective correspondence with $\lambda\mu$-terms, that freshness captures the concept of free variables/names, and that swapping is equivalent to bijective renaming.

The standard approach to typechecking $\lambda\mu$-terms is to use two contexts, Γ and Δ, for variable- and continuation-bindings respectively. The typechecking rules from the previous section may be adapted by adding Δ-contexts and adding new rules for the new syntax cases:

$$
\begin{array}{lcl}
tc & :: & \mathbf{list}\ (id \times ty) \times exp \times ty \times \mathbf{list}\ (con \times ty) \to o \\
tc(G, pass(X,E), bot, D) & :- & mem((X,T), D), tc(C, E, T, D). \\
tc(G, app(E,E'), bot, D) & :- & tc(C, E, bot, D), tc(C, E', T, D). \\
tc(G, mu(\langle \mathsf{a}\rangle E), T, D) & :- & \mathsf{a} \# D, tc(G, E, bot, [(\mathsf{a}, T)|D]).
\end{array}
$$

The following query illustrates the typechecking for the term $\lambda x.\mu\alpha.(x\ (\lambda y.[\alpha]y))$ whose principal type corresponds to the classical double negation law.

$$
\begin{array}{l}
?\text{–}\ tc([], lam(\langle \mathsf{x}\rangle mu(\langle \mathsf{a}\rangle app(var(\mathsf{x}), lam(\langle \mathsf{y}\rangle pass(\mathsf{a}, var(\mathsf{y})))))), T, []). \\
T = arr(arr(arr(T', bot), bot), T')
\end{array}
$$

Capture-avoiding substitution can be extended to $\lambda\mu$-terms easily. For replacement, we show the interesting cases for continuation applications and μ-abstractions:

$$
\begin{array}{lcll}
repl & :: & exp \times exp \times con \to exp. & \\
repl(pass(A, E'), E, A) & = & pass(A, app(repl(E', E, A), E)). & \\
repl(pass(B, E'), E, A) & = & pass(B, repl(E', E, A)) & :- \ A \# B. \\
repl(mu(\langle \mathsf{b}\rangle E'), E, A) & = & mu(\langle \mathsf{b}\rangle repl(E', E, A)) & :- \ \mathsf{b} \# (A, E).
\end{array}
$$

This approach seems quite different in flavor from a third-order HOAS encoding of the $\lambda\mu$-calculus due to Abel [2001]. There, $\mu\alpha.t$ is (essentially) encoded as $mu(\lambda^{\ulcorner}\alpha^{\urcorner}.^{\ulcorner}t^{\urcorner})$, where $mu : ((exp\ A \to exp\ bot) \to exp\ bot) \to exp\ A$, and $exp\ A$ is

Process terms $p ::= 0 \mid \tau.p \mid p|q \mid p+q \mid x(y).p \mid \overline{x}y.p \mid [x=y]p \mid [x \neq y]p \mid (x)p$
Actions $a ::= \tau \mid x(y) \mid \overline{x}y \mid \overline{x}(y)$

$chan$	: **name_type**.		
$proc$	: **type**.		
ina	: $proc$.		
tau	: $proc \to proc$.	act	: **type**.
par	: $proc \times proc \to proc$.	tau_a	: act.
sum	: $proc \times proc \to proc$.	in_a	: $chan \times chan \to act$.
in	: $chan \times \langle chan \rangle proc \to proc$.	$fout\_a$	: $chan \times chan \to act$.
out	: $chan \times chan \times proc \to proc$.	$bout\_a$	: $chan \times chan \to act$.
$match$	: $chan \times chan \times proc \to proc$.		
$mismatch$	: $chan \times chan \times proc \to proc$.		
res	: $\langle chan \rangle proc \to proc$.		

Fig. 6. The π-calculus: syntax and αProlog declarations

the type of terms of type A. Continuations are encoded as variables $\ulcorner\alpha\urcorner : exp\ A \to exp\ \bot$ and named terms $[\alpha]t$ are encoded as applications $\ulcorner\alpha\urcorner\ \ulcorner t\urcorner$. This encoding appears to let us define the β-reduction associated with μ elegantly as $app(mu(\lambda x : (exp\ A \to exp\ bot).F\ x))T \to_\beta F\ (\lambda y.y\ u)$. However, this encoding is not adequate, because there are terms of type $exp\ A \to exp\ \bot$ other than variables; for example, if $A = \bot$, the identity function inhabits this type. This means that further work is needed to restrict the encoding and recover adequacy, so it is more difficult to implement correct programs for such an encoding. Instead, Abel investigated an alternative second-order encoding which seems essentially the same as ours, and does not take advantage of built-in substitution for the replacement operation.

2.3 The π-calculus

The π-calculus is a calculus of concurrent, mobile processes. Its syntax (following Milner et al. [1992]) is described by the grammar rules shown in Figure 6. The symbols $x, y, \ldots$ are *channel names*. The inactive process 0 is inert. The $\tau.p$ process performs a *silent action* τ and then does p. Parallel composition is denoted $p|q$ and nondeterministic choice by $p+q$. The process $x(y).p$ inputs a channel name from x, binds it to y, and then does p. The process $\overline{x}y.p$ outputs y to x and then does p. The match operator $[x=y]p$ is p provided $x = y$, but is inactive if $x \neq y$. The restriction operator $(y)p$ restricts y to p. Parenthesized names (e.g. y in $x(y).p$ and $(y)p$) are binding, and $fn(p)$, $bn(p)$ and $n(p)$ denote the sets of free, bound, and all names occurring in p. Capture-avoiding renaming is written $t\{x/y\}$.

Milner et al.'s original operational semantics (shown in Figure 6, symmetric cases omitted) is a labeled transition system with relation $p \xrightarrow{a} q$ indicating "p steps to q by performing action a". Actions τ, $\overline{x}y$, $x(y)$, $\overline{x}(y)$ are referred to as *silent*, *free output*, *input*, and *bound output* actions respectively; the first two are called *free* and the second two are called *bound* actions. For an action a, $n(a)$ is the set of all names appearing in a, and $bn(a)$ is empty if a is a free action and is $\{y\}$ if a is a bound action $x(y)$ or $\overline{x}(y)$. Processes and actions can be encoded using the declarations shown in Figure 6.

Much of the complexity of the rules is due to the need to handle *scope extrusion*,

$$\tau.p \xrightarrow{\tau} p \qquad \frac{p \xrightarrow{a} p' \quad bn(a) \cap fn(q) = \varnothing}{p|q \xrightarrow{a} p'|q} \qquad \frac{p \xrightarrow{\overline{x}y} p' \quad q \xrightarrow{x(z)} q'}{p|q \xrightarrow{\tau} p'|q'\{y/z\}}$$

$$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \overline{x}y.p \xrightarrow{\overline{x}y} p \qquad \frac{w \notin fn((z)p)}{x(z).p \xrightarrow{x(w)} p\{w/z\}} \qquad \frac{p \xrightarrow{a} p'}{[x=x]p \xrightarrow{a} p'} \qquad \frac{(x \neq y) \quad p \xrightarrow{a} p'}{[x \neq y]p \xrightarrow{a} p'}$$

$$\frac{p \xrightarrow{\overline{x}(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \frac{p \xrightarrow{a} p' \quad y \notin n(a)}{(y)p \xrightarrow{a} (y)p'} \qquad \frac{p \xrightarrow{\overline{x}y} p' \quad y \neq x \quad w \notin fn((y)p)}{(y)p \xrightarrow{\overline{x}(w)} p'\{w/y\}}$$

```
ren_p                          :: proc × chan × chan → proc. (* definition omitted *)
safe                           :: act × pr → o. (* tests bn(A) ∩ fn(P) = ∅ *)
safe(tau_a, P).
safe(fout_a(X,Y), P).
safe(bout_a(X,Y), P)   :- Y # P.
safe(in_a(X,Y), P)     :- Y # P.
step                                                :: pr × act × pr → o. (* encodes p -a-> p' *)
step(tau(P), tau_a, P).
step(par(P,Q), A, par(P',Q))                        :- step(P, A, P'), safe(A, Q).
step(par(P,Q), tau_a, par(P', ren_p(Q', Y, Z)))     :- step(P, fout_a(X,Y), P'),
                                                       step(Q, in_a(X,Z), Q').
step(sum(P,Q), A, P')                               :- step(P, A, P').
step(out(X,Y,P), fout_a(X,Y), P).
step(in(X, ⟨z⟩P), in_a(X,W), ren_p(P,W,z))          :- W # ⟨z⟩P.
step(match(X,X,P), A, P')                           :- step(P, A, P').
step(match(X,Y,P), A, P')                           :- X # Y, step(P, A, P').
step(par(P,Q), tau_a, res(⟨z⟩par(P',Q')))           :- step(P, bout_a(X,z), P'),
                                                       step(Q, in_a(X,z), Q').
step(res(⟨y⟩P), A, res(⟨y⟩P'))                      :- y # A, step(P, A, P').
step(res(⟨y⟩P), bout_a(X,W), ren_p(P',W,y))         :- step(P, fout_a(X,y), P'), y # X,
                                                       W # ⟨y⟩P.
```

Fig. 7. π-calculus transitions and αProlog implementation

which occurs when restricted names "escape" their scope because of communication. In $((x)\overline{a}x.p)|(a(z).z(x).0) \xrightarrow{\tau} (x')(p|x'(x).0))$, for example, it is necessary to "freshen" x to x' in order to avoid capturing the free x in $a(z).z(x).0$. Bound output actions are used to lift the scope of an escaping name out to the point where it is received. The rules can be translated directly into αProlog (see Figure 7). The function $ren\_p(P, Y, X)$ performing capture-avoiding renaming is not shown, but easy to define.

We can check that this implementation of the operational semantics produces correct answers for the following queries:

```
?– step(res(⟨x⟩par(res(⟨y⟩out(x,y,ina)), in(x, ⟨z⟩out(z,x,ina)))), A, P).
A = tau_a, P = res(⟨y58⟩res(⟨z643⟩par(ina, out(z643, y58, ina))))
?– step(res(⟨x⟩out(x,y,ina)), A, P).
No.
```

This αProlog session shows that $(x)((y)\overline{x}y.0 \mid x(y).\overline{y}x.0) \xrightarrow{\tau} (x)(y)(0 \mid \overline{y}x.0)$, but $(x)(x(y).0)$ cannot make any transition. Moreover, the answer to the first query is unique (up to renaming).

Röckl [2001] and Gabbay [2003] have also considered encodings of the π-calculus using nominal abstract syntax. Röckl considered only modeling the syntax of terms up to α-equivalence using swapping, whereas Gabbay went further, encoding transitions and the bisimulation relation and proving basic properties thereof. By [Gabbay 2003, Thm 4.5], Gabbay's version of the π-calculus is equivalent to our conventional representation. In fact, Gabbay's presentation is a bit simpler to express in αProlog, but we have chosen Milner et al.'s original presentation to emphasize that informal "paper" presentations (even for fairly complicated calculi) can be translated directly to αProlog programs.

2.3.1 *Dyadic π-calculus.* The polyadic π-calculus adds to the π-calculus the ability to send and receive n-tuples of names, not just single names. It is a useful intermediate stage for translations form other languages (such as the λ-calculus, object calculi, or the ambient calculus) to the pure π-calculus. We can easily define a special case of dyadic π-terms (that can send and receive pairs of names) in αProlog:

$$
\begin{array}{lcl}
in2 & : & chan \times \langle chan \rangle \langle chan \rangle proc \to proc. \\
out2 & : & chan \times chan \times chan \times proc \to proc. \\
 & & \\
unpoly & :: & proc \to proc. \\
unpoly(out2(C,X,Y,P)) & = & res(\langle \mathsf{z} \rangle out(C, \mathsf{z}, out(\mathsf{z}, X, out(\mathsf{z}, Y, unpoly(P))))) \\
 & :- & \mathsf{z} \# (C,X,Y,P). \\
unpoly(in2(C, \langle \mathsf{x} \rangle \langle \mathsf{y} \rangle P)) & = & in(C, \langle \mathsf{z} \rangle in(\mathsf{z}, \langle \mathsf{x} \rangle in(\mathsf{z}, \langle \mathsf{y} \rangle unpoly(P)))) \\
 & :- & \mathsf{z} \# (C,P).
\end{array}
$$

2.3.2 *True polyadicity.* It is somewhat awkward to work with the $in2$ and $out2$ constructors. Ideally, we would prefer to be able to send an arbitrary n-tuple (i.e., list) of names along a channel. For output, this is no problem: we can easily change $out2$ to

$$out^* : chan \times \mathbf{list}\ chan \times proc \to proc$$

and modify *unpoly* appropriately. However, for inputs, we need to be able to bind a *list* of names

$$in^* : chan \times \langle \mathbf{list}\ chan \rangle proc \to proc$$

This would permit us to deal with in^* as follows:

$$
\begin{array}{lcl}
in^* & : & (chan, \langle \mathbf{list}\ chan \rangle proc) \to proc. \\
 & & \\
unpoly\_in & :: & chan \to \mathbf{list}\ chan \to proc \to proc \\
unpoly\_in\ Z\ []\ P & = & P. \\
unpoly\_in\ Z\ (X :: Xs)\ P & = & in(Z, \langle X \rangle unpoly\_in\ Z\ Xs\ P). \\
 & & \\
unpoly & :: & proc \to proc. \\
unpoly(in^*(C, \langle L \rangle P)) & = & in(C, \langle \mathsf{z} \rangle unpoly\_in\ \mathsf{z}\ L\ (unpoly\ P)) :- \mathsf{z} \# P.
\end{array}
$$

This behavior can be simulated using a user-defined type $listAbs$: **name_type** $\to$

type $\to$ **type** having constructors

$$\begin{array}{ll} nilAbs & : \beta \to listAbs\ \alpha\ \beta \\ consAbs & : \langle\alpha\rangle listAbs\ \alpha\ \beta \to listAbs\ \alpha\ \beta \end{array}$$

but programming with polyadic π-terms in this way is awkward. Dealing with more general binding structures "natively" in αProlog (as has been done in functional programming settings such as FreshML [Shinwell et al. 2003], Cαml [Pottier 2005], and FreshLib [Cheney 2005c]) is the subject of current research; however, the main challenges seem to be in designing appropriate general-purpose binding specification techniques and constraint solvers for generalized binding forms.

2.3.3 *Translation from λ-calculus to π-calculus.* Both call-by-value and call-by-name translations from the λ-calculus to (dyadic) π-calculus can be developed. We assume that the λ-calculus variables and π-calculus names coincide.

$$\begin{array}{lcl} cbv & :: & exp \times chan \to proc. \\ cbv(var(X), P) & = & out(P, X). \\ cbv(app(M, N), P) & = & res(\langle \mathsf{q}\rangle par(\ cbv(M, \mathsf{q}), \\ & & \qquad in(\mathsf{q}, \langle \mathsf{v}\rangle res(\langle \mathsf{r}\rangle par(\ cbv(N, \mathsf{r}), \\ & & \qquad\qquad in(\mathsf{r}, \langle \mathsf{w}\rangle out2(\mathsf{v}, \mathsf{w}, P, ina)))))))) \\ cbv(lam(\langle \mathsf{x}\rangle M), P) & = & res(\langle \mathsf{y}\rangle out(P, \mathsf{y}, rep(in2(\mathsf{y}, \langle \mathsf{x}\rangle\langle \mathsf{q}\rangle cbv(M, \mathsf{q}))))). \end{array}$$

This can be seen to be equivalent to an informal definition (paraphrasing [Sangiorgi and Walker 2001, Table 15.2]):

$$\begin{array}{rcl} \mathcal{V}[\![x]\!]p & = & \overline{p}x \\ \mathcal{V}[\![M\ N]\!]p & = & (q)\,(\mathcal{V}[\![M]\!]q \mid q(v).(r)(\mathcal{V}[\![N]\!]r \mid r(w).\overline{v}\langle w, p\rangle)) \\ \mathcal{V}[\![\lambda x.M]\!]p & = & \overline{p}(y).!y(x, q).\mathcal{V}[\![M]\!]q \end{array}$$

3. NOMINAL LOGIC AND HERBRAND MODELS

In the previous section, we employed a concrete syntax for αProlog programs which is convenient for writing programs, but less convenient for defining the semantics and reasoning about programs. We take the view that αProlog programs are interpreted as theories in nominal logic, just as pure Prolog programs can be viewed as theories of first-order logic. Consequently, we will now adopt an abstract syntax for αProlog programs that is based on the syntax of nominal logic. Thus, instead of the concrete syntax

$$goal(X, \langle \mathsf{a}\rangle Y) :- subgoal1(X), subgoal2(\mathsf{a}, Y).$$

we use the more explicit, logically equivalent formula

$$И\mathsf{a}.\forall X, Y.subgoal1(X) \wedge subgoal2(\mathsf{a}, Y) \Rightarrow goal(X, \langle \mathsf{a}\rangle Y)$$

This correspondence between αProlog goals and program and nominal logic formulas and theories will now be made precise.

The syntax of nominal logic is shown in Figure 8. We assume fixed countable sets of *variables* $\mathcal{V}$ and *names* $\mathbb{A}$. A *language* $\mathcal{L}$ consists of a set of *data types* δ, *name types* ν, *constants* $c : \delta$, *function symbols* $f : \vec{\sigma} \to \delta$, and *relation symbols* $p : \vec{\sigma} \to o$. The novel term constructors include names $\mathsf{a} \in \mathbb{A}$, name-abstractions $\langle a\rangle t$ denoting α-equivalence classes, and name-swapping applications $(a\ b) \cdot t$. Atomic formulas

$$
\begin{array}{lrcl}
\text{(Types)} & \sigma & ::= & \nu \mid \delta \mid \langle\nu\rangle\sigma \\
\text{(Contexts)} & \Sigma & ::= & \cdot \mid \Sigma, X{:}\sigma \mid \Sigma\#\mathsf{a}{:}\nu \\
\text{(Terms)} & t & ::= & \mathsf{a} \mid c \mid f(\vec{t}) \mid x \mid (a\ b)\cdot t \mid \langle a\rangle t \\
\text{(Constraints)} & C & ::= & t \approx u \mid a \# t \mid C \wedge C' \mid \exists X{:}\sigma.C \mid \mathit{И}\mathsf{a}{:}\sigma.C \\
\text{(Formulas)} & \phi & ::= & \top \mid \bot \mid p(\vec{t}) \mid C \mid \phi \Rightarrow \psi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \forall X{:}\sigma.\phi \mid \exists X{:}\sigma.\phi \mid \mathit{И}\mathsf{a}{:}\nu.\phi
\end{array}
$$

Fig. 8. Syntax of nominal logic

$$
\frac{\mathsf{a}:\nu \in \Sigma}{\Sigma \vdash \mathsf{a}:\nu} \quad \frac{x:\sigma \in \Sigma}{\Sigma \vdash x:\sigma} \quad \frac{c:\delta \in \mathcal{L}}{\Sigma \vdash c:\delta} \quad \frac{f:\vec{\sigma}\to\delta \in \mathcal{L} \quad \Sigma \vdash \vec{t}:\vec{\sigma}}{\Sigma \vdash f(\vec{t}):\delta} \quad \frac{\Sigma \vdash a:\nu \quad \Sigma \vdash t:\sigma}{\Sigma \vdash \langle a\rangle t : \langle\nu\rangle\sigma}
$$

$$
\frac{\Sigma \vdash a:\nu \quad \Sigma \vdash b:\nu \quad \Sigma \vdash t:\sigma}{\Sigma \vdash (a\ b)\cdot t:\sigma} \quad \frac{\Sigma \vdash t,u:\sigma}{\Sigma \vdash t \approx u : o} \quad \frac{\Sigma \vdash a:\nu \quad \Sigma \vdash t:\sigma}{\Sigma \vdash a \# t : o}
$$

$$
\frac{}{\Sigma \vdash \top, \bot : o} \quad \frac{\Sigma \vdash \phi,\psi : o}{\Sigma \vdash \phi \wedge \psi, \phi \vee \psi, \phi \Rightarrow \psi : o} \quad \frac{\Sigma, X{:}\sigma \vdash \phi : o}{\Sigma \vdash \forall X{:}\sigma.\phi, \exists X{:}\sigma.\phi : o} \quad \frac{\Sigma\#\mathsf{a}{:}\nu \vdash \phi : o}{\Sigma \vdash \mathit{И}\mathsf{a}{:}\nu.\phi : o}
$$

Fig. 9. Well-formedness for nominal terms and formulas

$$
\begin{array}{lll}
(\mathsf{a}\ \mathsf{b})\cdot\mathsf{a} = \mathsf{b} & & (\mathsf{a}\ \mathsf{b})\cdot c = c \\
(\mathsf{a}\ \mathsf{b})\cdot\mathsf{b} = \mathsf{a} & & (\mathsf{a}\ \mathsf{b})\cdot f(\vec{t}) = f((\mathsf{a}\ \mathsf{b})\cdot\vec{t}) \\
(\mathsf{a}\ \mathsf{b})\cdot\mathsf{a}' = \mathsf{a}' & (\mathsf{a} \neq \mathsf{a}' \neq \mathsf{b}) & (\mathsf{a}\ \mathsf{b})\cdot\langle\mathsf{a}'\rangle t = \langle(\mathsf{a}\ \mathsf{b})\cdot\mathsf{a}'\rangle(\mathsf{a}\ \mathsf{b})\cdot t
\end{array}
$$

$$
\frac{(\mathsf{a} \neq \mathsf{b})}{\vDash \mathsf{a} \# \mathsf{b}} \quad \frac{}{\vDash \mathsf{a} \# c} \quad \frac{\bigwedge_{i=1}^n \vDash \mathsf{a} \# t_i}{\vDash \mathsf{a} \# f(t_1^n)} \quad \frac{\vDash \mathsf{a} \# \mathsf{b} \quad \vDash \mathsf{a} \# t}{\vDash \mathsf{a} \# \langle\mathsf{b}\rangle t} \quad \frac{}{\vDash \mathsf{a} \# \langle\mathsf{a}\rangle t}
$$

$$
\frac{}{\vDash \mathsf{a} \approx \mathsf{a}} \quad \frac{}{\vDash c \approx c} \quad \frac{\bigwedge_{i=1}^n \vDash t_i \approx u_i}{\vDash f(t_1^n) \approx f(u_1^n)} \quad \frac{\vDash t \approx u}{\vDash \langle\mathsf{a}\rangle t \approx \langle\mathsf{a}\rangle u} \quad \frac{\vDash \mathsf{a} \# u \quad \vDash t \approx (\mathsf{a}\ \mathsf{b})\cdot u}{\vDash \langle\mathsf{a}\rangle t \approx \langle\mathsf{b}\rangle u}
$$

Fig. 10. Swapping, freshness, and equality for ground nominal terms

include freshness $a \# t$ and equality $t \approx u$. Well-formedness is defined for terms and formulas in Figure 9. Context bindings include ordinary typed variables $\Sigma, X{:}\sigma$ and name-typed names $\Sigma\#\mathsf{a}{:}\nu$. We write o for the type of propositions; quantification over types mentioning o is not allowed.

Figure 10 defines the swapping, freshness, and equality operations on ground terms. Swapping exchanges two syntactic occurrences of a name in a term (including occurrences such as a in $\langle\mathsf{a}\rangle t$.) The freshness relation defines what it means for a name to be "not free in" (or *fresh* for) a term. Intuitively, a name a is fresh for a term t (that is, $\mathsf{a} \# t$) if t possesses no occurrences of a unenclosed by an abstraction of a. Finally, the equality relation on nominal terms is defined using freshness and swapping. The only interesting cases are for abstractions; the second rule for abstractions is equivalent to more standard forms of α-renaming, as has been shown elsewhere [Gabbay and Pitts 2002; Pitts 2003].

We sometimes refer to the set of "free" names of a term $\mathit{supp}(t) = \mathbb{A} - \{\mathsf{a} \mid \mathsf{a} \# t\}$ as its *support*. Also, swapping and support are extended to formulas by setting $(\mathsf{a}\ \mathsf{b})\cdot QX.\phi[X] = QX.(\mathsf{a}\ \mathsf{b})\cdot\phi[X]$ for $Q \in \{\forall, \exists\}$ and $(\mathsf{a}\ \mathsf{b})\cdot \mathit{И}\mathsf{a}'.\phi = \mathit{И}\mathsf{a}'.(\mathsf{a}\ \mathsf{b})\cdot\phi$, provided $\mathsf{a}' \notin \{\mathsf{a}, \mathsf{b}\}$; thus, using α-renaming, we have $(\mathsf{a}\ \mathsf{b})\cdot\forall X.\mathit{И}\mathsf{a}.p(\mathsf{a}, \mathsf{b}, X) = \mathit{И}\mathsf{a}'.\forall X.p(\mathsf{a}', \mathsf{a}, X)$. Likewise, swapping can be extended to sets of terms or formulas by setting $(\mathsf{a}\ \mathsf{b})\cdot S = \{(\mathsf{a}\ \mathsf{b})\cdot t \mid t \in S\}$.

For the purposes of this paper, it suffices to restrict attention to *term models*

$$
\begin{array}{lcl@{\qquad}lcl}
\mathcal{H} \vDash \top & & & \mathcal{H} \vDash \phi \vee \psi & \iff & \mathcal{H} \vDash \phi \text{ or } \mathcal{H} \vDash \psi \\
\mathcal{H} \nvDash \bot & & & \mathcal{H} \vDash \phi \Rightarrow \psi & \iff & \mathcal{H} \vDash \phi \text{ implies } \mathcal{H} \vDash \psi \\
\mathcal{H} \vDash A & \iff & A \in \mathcal{H} & \mathcal{H} \vDash \forall X{:}\sigma.\phi & \iff & \text{for all } t : \sigma,\ \mathcal{H} \vDash \phi[t/X] \\
\mathcal{H} \vDash t \approx u & \iff & \vDash t \approx u & \mathcal{H} \vDash \exists X{:}\sigma.\phi & \iff & \text{for some } t : \sigma,\ \mathcal{H} \vDash \phi[t/X] \\
\mathcal{H} \vDash a \# u & \iff & \vDash a \# u & \mathcal{H} \vDash \mathsf{И}\mathsf{a}{:}\nu.\phi & \iff & \text{for all } \mathsf{b} : \nu \notin \mathrm{supp}(\mathsf{И}\mathsf{a}{:}\nu.\phi), \\
\mathcal{H} \vDash \phi \wedge \psi & \iff & \mathcal{H} \vDash \phi \text{ and } \mathcal{H} \vDash \psi & & & \mathcal{H} \vDash (\mathsf{b}\ \mathsf{a}) \cdot \phi.
\end{array}
$$

Fig. 11. Term model semantics of nominal logic

of nominal logic in which the domain elements are nominal terms with equality and freshness defined as in Figure 10. We write $B_{\mathcal{L}}$ for the *Herbrand base*, that is, the set of all ground instances of user-defined predicates p. We view an Herbrand model $\mathcal{H}$ as a subset of $B_{\mathcal{L}}$ that is *equivariant*, or closed under swapping (that is, $\mathcal{H} \subseteq (\mathsf{a}\ \mathsf{b}) \cdot \mathcal{H}$ for any a, b.) The semantics of nominal logic formulas over term models is defined as shown in Figure 11. The only nonstandard case is that for $\mathsf{И}$. This definition of the semantics of $\mathsf{И}$ has a dual form:

LEMMA 3.1. *The following are equivalent:*

(1) $\mathcal{H} \vDash \mathsf{И}\mathsf{a}.\phi$.

(2) $\mathcal{H} \vDash (\mathsf{a}\ \mathsf{b}) \cdot \phi$ *for some* $\mathsf{b} \notin \mathit{supp}(\mathsf{И}\mathsf{a}.\phi)$.

(3) $\mathcal{H} \vDash (\mathsf{a}\ \mathsf{b}) \cdot \phi$ *for every* $\mathsf{b} \notin \mathit{supp}(\mathsf{И}\mathsf{a}.\phi)$.

PROOF. (1) and (3) are equivalent by definition. (2) and (3) are equivalent because

$$\exists a.a \# \vec{x} \wedge \phi(a, \vec{x}) \iff \forall a.a \# \vec{x} \Rightarrow \phi(a, \vec{x})$$

is a theorem of nominal logic for any ϕ such that $FV(\phi) \subseteq \{a, \vec{x}\}$ [Pitts 2003, Prop. 4]. □

We define ground substitutions θ as functions from $\mathcal{V}$ to ground terms. Given a context Σ, we say that a ground substitution θ *satisfies* Σ (written $\theta : \Sigma$) when

$$\frac{}{\cdot : \cdot} \qquad \frac{\theta : \Sigma}{\theta, x \mapsto v : \Sigma, x} \qquad \frac{\mathsf{a} \# \theta \quad \theta : \Sigma}{\theta : \Sigma \# \mathsf{a}}$$

For example, $[X \mapsto \mathsf{a}, Y \mapsto \mathsf{b}]$ satisfies $\Sigma = \mathsf{a}, X\#\mathsf{b}, Y$ but not $X, Y\#\mathsf{a}\#\mathsf{b}$.

We generalize the satisfiability judgments as follows. Given sets of formulas Γ, Δ, we write

—$\mathcal{H} \vDash \Gamma$ (for Γ closed) to indicate that $\mathcal{H} \vDash \phi$ for each $\phi \in \Gamma$

—$\Gamma \vDash \Delta$ (for Γ, Δ closed) to indicate that $\mathcal{H} \vDash \Gamma$ implies $\mathcal{H} \vDash \Delta$

—$\Sigma : \theta \vDash \phi$ (for $\theta : \Sigma$) to indicate that $\mathcal{H} \vDash \theta(\phi)$.

—$\Sigma : \Gamma, \theta \vDash \Delta$ to indicate that $\theta(\Gamma) \vDash \theta(\Delta)$

—$\Sigma : \Gamma \vDash \Delta$ to indicate that $\Sigma : \Gamma, \theta \vDash \Delta$ for every $\theta : \Sigma$

—$\forall\Sigma[\phi]$ (or $\exists\Sigma[\phi]$) for the formula obtained by $\forall$-quantifying (or $\exists$-quantifying) all variables and $\mathsf{И}$-quantifying all names in Σ.

Note that, for example, $X\#\mathsf{a} : \cdot \vDash \mathsf{a} \# X$ but $\mathsf{a}, X : \cdot \nvDash \mathsf{a} \# X$.

We enumerate a number of basic properties of satisfiability, most of which are standard.

LEMMA 3.2. *If* $\Sigma : \Gamma \vDash \phi$ *and* $\Sigma : \Gamma, \phi \vDash \psi$ *then* $\Sigma : \Gamma \vDash \psi$.

LEMMA 3.3. *If* $\Sigma : \Gamma \vDash \exists X.\psi$ *and* $\Sigma, X : \Gamma, \psi \vDash \phi$ *hold then* $\Sigma : \Gamma \vDash \exists X.\phi$ *holds.*

LEMMA 3.4. *If* $\Sigma : \Gamma \vDash \mathsf{И}\mathsf{a}.\psi$ *and* $\Sigma\#\mathsf{a} : \Gamma, \psi \vDash \phi$ *hold then* $\Sigma : \Gamma \vDash \mathsf{И}\mathsf{a}.\phi$ *holds.*

LEMMA 3.5. *If* $\Sigma : \Gamma, \psi_i \vDash \phi$ *then* $\Sigma : \Gamma, \psi_1 \wedge \psi_2 \vDash \phi$.

LEMMA 3.6. *If* $\Sigma : \Gamma \vDash \psi_1$ *and* $\Sigma : \Gamma, \psi_2 \vDash \phi$ *then* $\Sigma : \Gamma, \psi_1 \Rightarrow \psi_2 \vDash \phi$.

LEMMA 3.7. *If* $\Sigma, X : \Gamma, \psi, \theta[X \mapsto t] \vDash \phi$ *and* X *does not appear in* Γ, ϕ *then* $\Sigma : \Gamma, \forall X.\psi, \theta \vDash \phi$.

LEMMA 3.8. *If* $\Sigma\#\mathsf{a} : \Gamma, \psi \vDash \phi$ *for some* a *not appearing in* Γ, ϕ *then* $\Sigma : \Gamma, \mathsf{И}\mathsf{a}.\psi \vDash \phi$.

We define the *nominal Horn goal formulas* G and *nominal Horn program clauses* D as follows:

$$G ::= \top \mid A \mid C \mid G \wedge G' \mid G \vee G' \mid \exists X.G \mid \mathsf{И}\mathsf{a}.G$$
$$D ::= \top \mid A \mid D \wedge D' \mid G \Rightarrow D \mid \forall X.D \mid \mathsf{И}\mathsf{a}.D$$

A nominal logic program is a set of closed clause formulas D.

Remark 3.9. The current implementation of αProlog includes support for negation-as-failure, conditionals, and the "cut" goal, which are not included in the above core language. We shall investigate only the semantics of "pure" αProlog programs. We believe that our results can be extended to negation-as-failure following [Jaffar et al. 1998] relatively easily. Also, we consider only monomorphic programs. It also appears straightforward to extend our results to general constraint domains or functional constraint logic programming.

4. SEMANTICS

So far, we have motivated αProlog purely in intuitive terms, arguing that αProlog concepts such as freshness and name-abstraction behave as they do "on paper". However, in order to prove the correctness of the example programs we have considered, it is important to provide a semantic foundation for reasoning about such programs. We shall investigate model-theoretic, proof-theoretic, and operational semantics for nominal logic programs.

Classical model-theoretic semantics for logic programming [van Emden and Kowalski 1976; Lloyd 1987] defines the meaning of a program as a Herbrand model constructed as the least fixed point of a continuous operator. We take for granted the theory of Herbrand models for nominal logic introduced in the previous section (full details are presented in [Cheney 2006a]). We then define an appropriate least fixed point semantics for nominal logic programs and prove that the least fixed point model and the least Herbrand model coincide.

While model-theoretic semantics is convenient for relating formal and informal systems, it is not as useful for implementation purposes. Instead, syntactic techniques based on proof theory are more appropriate because they provide a declarative reading of connectives as proof search operations in constructive logic. Miller et al. [1991] introduced the concept of *uniform proof*; a collection of program clauses

and goal formulas is considered an *abstract logic programming language* if goal-directed proof search is complete with respect to the underlying logic. Accordingly, we introduce a proof theory for a fragment of intuitionistic nominal logic which performs goal-directed proof search (decomposes complex goals to simple atomic formulas) and focused resolution (searches systematically for proofs of atomic formulas based on the syntax of program clauses). We prove the soundness and completeness of this system with respect to the model-theoretic semantics.

Finally, we consider the operational semantics of nominal logic programs at an abstract level. The proof theoretic semantics contains a number of "don't-know" nondeterministic choices. We provide an operational semantics (following the semantics of constraint logic programming [Jaffar et al. 1998; Darlington and Guo 1994; Leach et al. 2001]) which delays these choices as long as possible, and closely models the behavior of an interpreter.

Along the way we prove appropriate soundness and completeness results relating the model-theoretic, proof-theoretic, and operational semantics. These results ensure the correctness of a low-level interpreter based on the operational semantics relative to the high-level approaches, and provide a rich array of tools for analyzing the behavior of nominal logic programs. The model-theoretic semantics is especially useful for relating informal systems with nominal logic programs, while the proof-theoretic semantics is convenient for proving behavioral properties of programs and program transformations. We shall consider such applications in Section 5.

4.1 Model-theoretic semantics

In this section we define the model-theoretic semantics of nominal logic programs. We show that least Herbrand models exist for nominal Horn clause programs and that the least Herbrand model is the least fixed point of an appropriate continuous one-step deduction operator, following Lloyd [1987]. This section also relies on standard definitions and concepts from lattice theory [Davey and Priestley 2002].

Although the overall structure of our proof follows Lloyd, it differs in some important technical details. Most importantly, we do not assume that clauses have been normalized to the form $A :- G$. Instead, all definitions and proofs are by induction over the structure of goals and program clauses. This is advantageous because it permits a much cleaner treatment of each logical connective independently of the others; this is especially helpful when considering the new cases arising for the И-quantifier, and when relating the model-theoretic semantics to the proof-theoretic and operational semantics.

4.1.1 *Least Herbrand Models.* It is a well-known fact that least Herbrand models exist for Horn clause theories in first-order logic. This is also true for nominal Horn clause theories. We rely on a previous development of Herbrand model theory for nominal logic [Cheney 2006a], culminating in the completeness of Herbrand models for Horn clause theories:

THEOREM 4.1 COMPLETENESS OF NOMINAL HERBRAND MODELS. *A collection of program clauses is satisfiable in nominal logic if and only if it has an Herbrand model.*

PROOF. We note without proof that we can prenex-normalize all ∃ and И quan-

tifiers in goals in D-formulas out to the top level as $\forall$ and $\mathsf{И}$ quantifiers respectively. Then a collection of normalized D-formulas is a $\mathsf{И}\forall$-theory in the sense of [Cheney 2006a, Theorem 6.17], so has a model iff it has an Herbrand model. □

LEMMA 4.2. *Let Δ be a program and $\mathcal{M}$ a nonempty set of Herbrand models of Δ. Then $\mathcal{H} = \bigcap \mathcal{M}$ is also an Herbrand model of Δ.*

PROOF. We first note that the intersection of a collection of equivariant sets is still equivariant, so $\mathcal{H}$ is an Herbrand model. To prove it models Δ, we show by mutual induction that

(1) For any program clause D, if $\forall M \in \mathcal{M}.M \vDash D$ then $\mathcal{H} \vDash D$; and
(2) For any goal formula G, if $\mathcal{H} \vDash G$ then $\forall M \in \mathcal{M}.M \vDash G$.

All the cases are standard except for $\mathsf{И}\mathsf{a}.G$ and $\mathsf{И}\mathsf{a}.D$. If $\forall M \in \mathcal{M}.M \vDash \mathsf{И}\mathsf{a}.D$ then for each M, $M \vDash (\mathsf{b}\ \mathsf{a}) \cdot D$ for all b not in $\mathrm{supp}(\mathsf{И}\mathsf{a}.D)$. Choose a $\mathsf{b} \notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D)$ such that $\forall M \in \mathcal{M}.M \vDash (\mathsf{b}\ \mathsf{a}) \cdot D$. Appealing to the induction hypothesis, we obtain $\mathcal{H} \vDash (\mathsf{b}\ \mathsf{a}) \cdot D$. By Lemma 3.1, it follows that $\mathcal{H} \vDash \mathsf{И}\mathsf{a}.D$. The case for $\mathsf{И}\mathsf{a}.G$ is similar (but simpler). □

An immediate consequence is that a least Herbrand model $\mathcal{H}_\Delta = \bigcap\{\mathcal{H} \mid \mathcal{H} \vDash \Delta\}$ exists for any nominal Horn theory Δ. Moreover, $\mathcal{H}_\Delta$ consists of all ground atoms entailed by Δ, as we now show.

THEOREM 4.3. *Let Δ be a program. Then $\mathcal{H}_\Delta = \{A \in B_\mathcal{L} \mid \Delta \vDash A\}$.*

PROOF. If $A \in \mathcal{H}_\Delta$, then A is valid in every Herbrand model of Δ, so by Theorem 4.1, A is valid in every model of Δ. Conversely, if $\Delta \vDash A$ then since $\mathcal{H}_\Delta \vDash \Delta$ we have $\mathcal{H}_\Delta \vDash A$; thus $A \in \mathcal{H}_\Delta$. □

4.1.2 *Fixed Point Semantics.* Classical fixed point theorems assert the existence of a fixed point. However, to ensure that the fixed point of an operator on nominal Herbrand models is still an Herbrand model we need an additional constraint: we require that the operator is also equivariant, in the following sense.

Definition 4.4. A set operator $T : \mathcal{P}(B_\mathcal{L}) \to \mathcal{P}(B_\mathcal{L})$ is called *equivariant* if $(\mathsf{a}\ \mathsf{b}) \cdot T(S) = T((\mathsf{a}\ \mathsf{b}) \cdot S)$.

THEOREM 4.5. *Suppose $T : \mathcal{P}(B_\mathcal{L}) \to \mathcal{P}(B_\mathcal{L})$ is equivariant and monotone. Then $\mathrm{lfp}(T) = \bigcap\{S \in \mathcal{P}(B_\mathcal{L}) \mid T(S) \subseteq S\}$ is the least fixed point of T and is equivariant. If, in addition, T is continuous, then $\mathrm{lfp}(T) = T^\omega = \bigcup_{i=0}^{\omega} T^i(\varnothing)$.*

PROOF. By the Knaster-Tarski fixed-point theorem, $\mathrm{lfp}(T)$ is the least fixed point of T. To show that $\mathrm{lfp}(T)$ is equivariant, it suffices to show that $A \in \mathrm{lfp}(T) \implies (\mathsf{a}\ \mathsf{b}) \cdot A \in \mathrm{lfp}(T)$. Let a, b be given and assume $A \in \mathrm{lfp}(T)$. Then for any pre-fixed point S of T (satisfying $T(S) \subseteq S$), we have $A \in S$. Let such an S be given. Note that $T((\mathsf{a}\ \mathsf{b}) \cdot S) = (\mathsf{a}\ \mathsf{b}) \cdot T(S) \subseteq (\mathsf{a}\ \mathsf{b}) \cdot S$, so $(\mathsf{a}\ \mathsf{b}) \cdot S$ is also a pre-fixed point of T. Hence $A \in (\mathsf{a}\ b) \cdot S$ so $(\mathsf{a}\ b) \cdot A \in (\mathsf{a}\ b) \cdot (\mathsf{a}\ b) \cdot S = S$. Since S was an arbitrary pre-fixed point, it follows that $(\mathsf{a}\ \mathsf{b}) \cdot A \in \mathrm{lfp}(T)$, as desired.

The second part follows immediately from Kleene's fixed point theorem. □

Definition 4.6. Let S be an Herbrand interpretation and D a closed program clause. The *one-step deduction operator* $T_D : \mathcal{P}(B_\mathcal{L}) \to \mathcal{P}(B_\mathcal{L})$ is defined as follows:

$$\begin{array}{rcl} T_\top(S) & = & S \\ T_A(S) & = & S \cup \{A\} \\ T_{D_1 \wedge D_2}(S) & = & T_{D_1}(S) \cup T_{D_2}(S) \\ T_{G \Rightarrow D}(S) & = & \begin{cases} T_D(S) & \text{if } S \vDash G \\ S & \text{otherwise} \end{cases} \\ T_{\forall X:\sigma.D}(S) & = & \bigcup_{t:\sigma} T_{D[t/X]}(S) \\ T_{\mathsf{И}\mathsf{a}:\nu.D}(S) & = & \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D)} T_{(\mathsf{a}\ \mathsf{b})\cdot D}(S) \end{array}$$

We define T_Δ as $T_{D_1 \wedge \cdots \wedge D_n}$ provided $\Delta = \{D_1, \ldots, D_n\}$ and each D_i is closed.

Remark 4.7. Many prior expositions of the model-theoretic semantics of logic programs treat "open" Horn clauses $A :- B_1, \ldots, B_n$ as the basic units of computation. For example, the one-step deduction operator is usually formulated as

$$T(S) = \{\theta(A) \mid \exists (A :- B_1, \ldots, B_n \in P), \theta.S \vDash \theta(B_1), \ldots, \theta(B_n)\}$$

This definition is not straightforward to extend to nominal logic programming because of the presence of the $\mathsf{И}$-quantifier. Although it can be done [Cheney 2004b, Chapter 6], the resulting model-theoretic semantics is difficult to relate to the proof-theoretic and operational semantics. Instead, we prefer to define T by induction on the structure of program clauses. This necessitates reorganizing our proofs, but the resulting argument is more modular with respect to extensions based on connectives.

LEMMA 4.8. *For any program Δ, T_Δ is monotone and continuous.*

PROOF. We prove by induction on the structure of D that T_D has the above properties. Monotonicity is straightforward. For continuity, let $S_0, S_1, \ldots,$ be an ω-chain of subsets of $B_\mathcal{L}$. The cases for $\top, \wedge, \Rightarrow, \forall,$ and atomic formulas follow standard arguments (see Appendix A). Suppose $D = \mathsf{И}\mathsf{a}.D'$. Then we have

$$\begin{array}{rcll} T_{\mathsf{И}\mathsf{a}.D'}(\bigcup_i S_i) & = & \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\bigcup_i S_i) & \text{Definition} \\ & = & \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} \bigcup_i T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(S_i) & \text{Induction hyp.} \\ & = & \bigcup_i \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(S_i) & \text{Unions commute} \\ & = & \bigcup_i T_{\mathsf{И}\mathsf{a}.D'}(S_i) & \text{Definition} \end{array}$$

This completes the proof. □

LEMMA 4.9. *For any $\mathsf{a}, \mathsf{b} \in \mathbb{A}$, $(\mathsf{a}\ \mathsf{b}) \cdot T_D(S) = T_{(\mathsf{a}\ \mathsf{b})\cdot D}((\mathsf{a}\ \mathsf{b})\cdot S)$. In particular, if Δ is a closed program with $FV(\Delta) = \mathrm{supp}(\Delta) = \varnothing$, then T_Δ is equivariant.*

PROOF. The proof is by induction on the structure of D. The cases for $\top, A, \wedge$ are straightforward (see Appendix A); for $\Rightarrow$ we need the easy observation that $S \vDash G \iff (\mathsf{a}\ \mathsf{b})\cdot S \vDash (\mathsf{a}\ \mathsf{b})\cdot G$. For $\forall X{:}\sigma.D$ formulas, observe that

$$\begin{array}{rcll} (\mathsf{a}\ \mathsf{b})\cdot T_{\forall X.D}(S) & = & (\mathsf{a}\ \mathsf{b})\cdot \bigcup_{t:\sigma} T_{D[t/X]}(S) & \text{Definition} \\ & = & \bigcup_{t:\sigma} (\mathsf{a}\ \mathsf{b}) \cdot T_{D[t/X]}(S) & \text{Swapping commutes with union} \\ & = & \bigcup_{t:\sigma} T_{((\mathsf{a}\ \mathsf{b})\cdot D)[(\mathsf{a}\ \mathsf{b})\cdot t/X]}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Induction hyp.} \\ & = & \bigcup_{u:\sigma} T_{((\mathsf{a}\ \mathsf{b})\cdot D)[u/X]}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Change of variables } (u = (\mathsf{a}\ \mathsf{b}) \cdot t) \\ & = & T_{(\mathsf{a}\ \mathsf{b})\cdot \forall X.D}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Definition.} \end{array}$$

For И, the argument is similar. □

LEMMA 4.10. *If $\mathcal{M}$ is a fixed point of T_Δ, then $\mathcal{M} \vDash \Delta$.*

PROOF. We first prove by induction on the structure of D that if $T_D(\mathcal{M}) = \mathcal{M}$ then $\mathcal{M} \vDash D$. We show only the case for И; the full proof is in Appendix A. For $D = \mathsf{И a}{:}\nu.D'$, note that $\mathcal{M} = T_{\mathsf{И a}.D'}(\mathcal{M}) = \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M})$ implies $T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) = \mathcal{M}$ for every fresh b. Hence by the induction hypothesis $\mathcal{M} \vDash (\mathsf{a}\ \mathsf{b}) \cdot D'$ for every fresh b; consequently $M \vDash \mathsf{И a}.D'$.

Since any program $\Delta = \{D_1, \ldots, D_n\}$ is equivalent to a D-formula $D_1 \wedge \cdots \wedge D_n$, the desired result follows immediately. □

LEMMA 4.11. *If $\mathcal{M} \vDash \Delta$ then $\mathcal{M}$ is a fixed point of T_Δ.*

PROOF. Since T_Δ is monotone it suffices to show that $\mathcal{M}$ is a pre-fixed point. We first prove that for any D, if $\mathcal{M} \vDash D$ then $T_D(\mathcal{M}) \subseteq \mathcal{M}$, by induction on the structure of D. We show only the case for И; other cases are in Appendix A. If $D = \mathsf{И a}{:}\nu.D'$, by assumption $\mathcal{M} \vDash \mathsf{И a}{:}\nu.D'$ so $\mathcal{M} \vDash (\mathsf{a}\ \mathsf{b}) \cdot D'$ for any $\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.D')$. By induction $T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) \subseteq \mathcal{M}$ for any $\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.D')$ so $\bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) \subseteq \mathcal{M}$.

To prove the lemma, take $\Delta = \{D_1, \ldots, D_n\}$ and $D = D_1 \wedge \cdots \wedge D_n$. If $\mathcal{M} \vDash \Delta$, then $\mathcal{M} \vDash D$, so $T_D(\mathcal{M}) \subseteq \mathcal{M}$, whence $T_\Delta(\mathcal{M}) \subseteq \mathcal{M}$. □

THEOREM 4.12. *$\mathcal{H}_\Delta = \mathrm{lfp}(T_\Delta) = T_\Delta^\omega$.*

PROOF. Clearly $T_\Delta^\omega = \mathrm{lfp}(T_\Delta)$ by Theorem 4.5. Moreover, by Lemma 4.11 and Lemma 4.10, the set of models of Δ equals the set of fixed points of T_Δ, so we must have $H_\Delta = \mathrm{lfp}(T_\Delta)$, since $\mathcal{H}_\Delta$ is the least model of Δ and $\mathrm{lfp}(T_\Delta)$ is the least fixed point of T_Δ. □

4.2 Proof-theoretic semantics

In proof-theoretic semantics, an approach due to Miller et al. [1991], well-behaved logic programming languages are characterized as those for which *uniform* (or *goal-directed*) proof search is complete. Uniform proofs are sequent calculus proofs in which right-decomposition rules are always used to decompose the goal before any other proof rules are considered. Proof-theoretic semantics has been extended to a variety of settings; most relevant here is work on constraint logic programming in a proof theoretic setting [Darlington and Guo 1994; Leach et al. 2001].

A uniform proof-theoretic approach to nominal logic programming was investigated by Gabbay and Cheney [2004] in the context of NL_{Seq}, an early sequent calculus formulation of nominal logic. However, this approach was unsatisfactory in some respects.

First, the underlying sequent calculus suggested an approach to proof-search for И-formulas quite unlike the intuitive "generate a fresh name and proceed" approach employed in αProlog. This problem has been addressed by an alternative sequent calculus for nominal logic called $NL^{\Rightarrow}$ [Cheney 2005d], in which the И-quantifier rules take a simpler form.

Second, some rules of NL_{seq} (and $NL^{\Rightarrow}$) are not goal-directed but cannot be permuted past the $\exists R$ and $\forall L$ rules. Instead, in these systems it appears necessary

$$\frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow C}\ con \qquad \frac{\Sigma : \Delta; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R$$

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2}\ \vee R_i \qquad \frac{}{\Sigma : \Delta; \nabla \Longrightarrow \top}\ \top R \qquad \frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \exists X{:}\sigma.G}\ \exists R$$

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \text{И}\mathsf{a}{:}\nu.G}\ \text{И}R \qquad \frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp \qquad \frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \qquad \frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A}\ \Rightarrow L$$

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A}\ \forall L \qquad \frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}:\nu.D} A}\ \text{И}L$$

Fig. 12. Uniform/focused proof search for intuitionistic nominal logic

to weaken the definition of uniform proof in order to permit applications of non-goal-directed rules before $\exists R$ and $\forall L$.

For example, $NL^{\Rightarrow}$ contains a *freshness rule* (F) that asserts that a fresh name can be introduced at any point in an argument:

$$\frac{\Sigma \# \mathsf{a} : \Gamma \Rightarrow \phi}{\Sigma : \Gamma \Rightarrow \phi}\ F \quad (\mathsf{a} \notin \Sigma)$$

Here, the judgment $\Sigma : \Gamma \Rightarrow \phi$ can be read as "For any valuation satisfying Σ, if all the formulas of Γ hold then ϕ holds." As the following partial derivation suggests, the goal formula $\text{И}\mathsf{a}.\exists x.\mathsf{a} \# x$ cannot be derived in $NL^{\Rightarrow}$ without using the "freshness rule" before $\exists R$, because otherwise there is no way to obtain a ground name b distinct from a with which to instantiate X:

$$\cfrac{\cfrac{\cfrac{\vdots}{\mathsf{a}\#\mathsf{b} : \cdot \Rightarrow \mathsf{a} \# \mathsf{b}}}{\mathsf{a}\#\mathsf{b} : \cdot \Rightarrow \exists X.\mathsf{a} \# X}\ \exists R}{\mathsf{a} : \cdot \Rightarrow \exists X.\mathsf{a} \# X}\ F}{\cdot : \cdot \Rightarrow \text{И}\mathsf{a}.\exists X.\mathsf{a} \# X}\ \text{И}R$$

We adopt a variation of $NL^{\Rightarrow}$ that also addresses the second problem: specifically, we define an "amalgamated" proof system $NL^{\Rightarrow}_{\models}$ that separates the term-level constraint-based reasoning from logical reasoning and proof search. This technique was employed by Darlington and Guo [1994] and further developed by Leach et al. [2001] in studying the semantics of constraint logic programs.

In this section we introduce the amalgamated proof system $NL^{\Rightarrow}_{\models}$ and relate it to the model-theoretic semantics in the previous section. We also introduce a second *residuated* proof system that explicates the process of reducing a goal to an answer constraint; this system forms an important link between the proof theory and the operational semantics in the next section. Residuated proof search corresponds to ordinary proof search in a natural way.

4.2.1 *The amalgamated system* $NL^{\Rightarrow}_{\models}$. The proof rules in Figure 12 describe a proof system that first proceeds by decomposing the goal to an atomic formula, which is then solved by refining a program clause. The uniform derivability judgment $\Sigma : \Delta; \nabla \Longrightarrow G$ indicates that G is derivable from Δ and ∇ in context Σ, while the focused proof judgment $\Sigma : \Delta; \nabla \xrightarrow{D} A$ indicates that atomic goal A is derivable from Δ and ∇ by refining the program clause D (using Δ to help solve any residual goals). The judgment $\Sigma : \nabla \models C$ is the ordinary constraint entailment relation defined in Section 3.

These rules are unusual in several important respects. First, the *hyp* rule requires solving a constraint of the form $A \sim B$, which we define as "there exists a permutation π such that $\pi \cdot A \approx B$". In contrast usually the hypothesis rule requires only that $A \approx A'$. Our rule accounts for the fact that equivalent atomic formulas may not be syntactically equal as nominal terms, but only equal modulo a permutation. Second, the proof system treats constraints specially, separating them into a context ∇. This is necessary because the role of constraints is quite different from that of program clauses: the former are used exclusively for constraint solving whereas the latter are used in backchaining. Third, the $И L, И R, \exists R$ and $\forall L$ rules are permitted to introduce a constraint on the witness name a or variable X rather than providing a witness term. This constraint-based treatment makes it possible to compartmentalize all reasoning about the constraint domain in the judgment $\Sigma : \nabla \models C$.

For example, the goal $И\mathsf{a}.\exists X.\mathsf{a} \# X$ has the following uniform derivation:

$$\dfrac{\Sigma\#\mathsf{a} : \nabla \models И\mathsf{a}.\top \qquad \dfrac{\dfrac{\Sigma\#\mathsf{a} : \nabla, \top \models \exists X.\mathsf{a} \# X \quad \Sigma\#\mathsf{a}, X : \Delta; \nabla, \top, \mathsf{a} \# X \Longrightarrow \mathsf{a} \# X}{\Sigma\#\mathsf{a} : \Delta; \nabla, \top \Longrightarrow \exists X.\mathsf{a} \# X}\ \exists R}{\Sigma : \Delta; \nabla \Longrightarrow И\mathsf{a}.\exists X.\mathsf{a} \# X}\ И R$$

since $\Sigma\#\mathsf{a} : \nabla \models \exists X.\mathsf{a} \# X$ is clearly valid for any ∇ (take X to be any ground name besides a).

We state without proof the following basic properties:

LEMMA 4.13.

(1) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *(or* $\Sigma : \Delta; \nabla \xrightarrow{D} A$*) and* Σ, Σ' *is a well-formed context then* $\Sigma, \Sigma' : \Delta; \nabla \Longrightarrow G$ *(or* $\Sigma, \Sigma' : \Delta; \nabla \xrightarrow{D} A$*).*

(2) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *(or* $\Sigma : \Delta; \nabla \xrightarrow{D} A$*) and* $\Delta \subseteq \Delta'$ *then* $\Sigma : \Delta, \Delta'; \nabla \Longrightarrow G$ *(or* $\Sigma : \Delta, \Delta'; \nabla \xrightarrow{D} A$*).*

(3) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *(or* $\Sigma : \Delta; \nabla \xrightarrow{D} A$*) and* $\Sigma : \nabla' \models \nabla$ *then* $\Sigma : \Delta; \nabla' \Longrightarrow G$ *(or* $\Sigma : \Delta; \nabla' \xrightarrow{D} A$*).*

We first show that the restricted system is sound with respect to the model-theoretic semantics.

THEOREM 4.14 (SOUNDNESS).

(1) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *is derivable then* $\Sigma : \Delta, \nabla \models G$.

(2) *If* $\Sigma : \Delta; \nabla \xrightarrow{D} G$ *is derivable then* $\Sigma : \Delta, D, \nabla \models G$.

PROOF. Induction on derivations. The only novel cases involve И.

Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \text{И}\mathsf{a}.G}\ \text{И}R$$

By induction we have that $\Sigma\#\mathsf{a} : \Delta, \nabla, C \vDash G$. Appealing to Lemma 3.4, we conclude $\Sigma : \Delta, \nabla \vDash \text{И}\mathsf{a}.G$.

Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

Then by induction hypothesis (2), we have that $\Sigma : \Delta, D, \nabla \vDash A$. Since $D \in \Delta$, clearly $\Sigma : \Delta \vDash D$ so we can deduce $\Sigma : \Delta, \nabla \vDash A$.

For the second part, proof is by induction on the derivation of $\Sigma : \Delta; \nabla \xrightarrow{D} G$. The interesting cases are *hyp* and $\text{И}L$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp$$

We need to show $\Sigma : \Delta, A', \nabla \vDash A$. To see this, suppose θ satisfies ∇ and $\mathcal{H}$ is an Herbrand model of $\Delta, \theta(A')$. Since $\Sigma : \nabla \vDash A' \sim A$, there must be a permutation π such that $\pi{\cdot}\theta(A') = \theta(A)$. Moreover, since $\mathcal{H} \vDash \theta(A')$, by the equivariance of $\mathcal{H}$ we also have $\mathcal{H} \vDash \pi{\cdot}\theta(A')$ so $\mathcal{H} \vDash \theta(A)$. Since θ and $\mathcal{H}$ were arbitrary, we conclude that $\Sigma : \Delta, A', \nabla \vDash A$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}.D} A}\ \text{И}L$$

By induction, we know that $\Sigma\#\mathsf{a} : \Delta, D, \nabla, C \vDash A$. Since $\Sigma : \nabla \vDash \text{И}\mathsf{a}.C$ it follows that $\Sigma\#\mathsf{a} : \Delta, D, \nabla, C \vDash A$, so by Lemma 3.2 we have $\Sigma\#\mathsf{a} : \Delta, D, \nabla \vDash A$. Moreover, by Lemma 3.8, we can conclude $\Sigma : \Delta, \text{И}\mathsf{a}.D, \nabla \vDash A$.

This completes the proof. □

We next show a restricted form of completeness relative to the model-theoretic semantics. Since the model-theoretic semantics is classical while the proof theory is constructive, it is too much to expect that classical completeness holds. For example, $A, B : \cdot \vDash A \approx B \vee A \# B$ is valid, but $A, B : \cdot; \cdot \Longrightarrow A \approx B \vee A \# B$ is not derivable (and indeed not intuitionistically valid). Instead, however, we can prove that any valuation θ that satisfies a goal G also satisfies a constraint which entails G.

PROPOSITION 4.15. *For any* $\Sigma, \Delta, G, D, i \geq 0$*:*

(1) If $\Sigma : T^i_\Delta, \theta \vDash G$ *then there exists* ∇ *such that* $\Sigma : \theta \vDash \nabla$ *and* $\Sigma : \Delta; \nabla \Longrightarrow G$ *is derivable.*

(2) *If* $\Sigma : T_{\theta(D)}(T^i_\Delta), \theta \vDash A$ *but* $\Sigma : T^i_\Delta, \theta \nvDash A$ *then there exists* ∇ *such that* $\Sigma : \theta \vDash \nabla$ *and* $\Sigma : \Delta; \nabla \xrightarrow{D} A$.

Proof. For the first part, proof is by induction on i and G; most cases are straightforward. We give two illustrative cases.

—If $G = A$ and $i > 0$, then there are two further cases. If $\Sigma : T^{i-1}_\Delta, \theta \vDash A$ then we use part (1) of the induction hypothesis. Otherwise $\Sigma : T^{i-1}_\Delta, \theta \nvDash A$. This implies that $\theta(A) \in T_\Delta(T^{i-1}_\Delta) = \bigcup_{D \in \Delta} T_D(T^{i-1}_\Delta)$, so we must have $\theta(A) \in T_D(T^{i-1}_\Delta)$ for some $D \in \Delta$. Observe that since D is closed, $\theta(D) = D$. Consequently $\Sigma : T_{\theta(D)}(T^{i-1}_\Delta), \theta \vDash A$ but $\Sigma : T^{i-1}_\Delta \nvDash A$, so induction hypothesis (2) applies and we can obtain a derivation of $\Sigma : \Delta; \nabla \xrightarrow{D} A$. The following derivation completes this case:

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel\ .$$

—If $G = \mathsf{И a}{:}\nu.G'$, assume without loss of generality $\mathsf{a} \notin \Sigma$. Then $\Sigma : T^i_\Delta, \theta \vDash \mathsf{И a}.G'$ implies $\Sigma\#\mathsf{a} : T^i_\Delta, \theta \vDash G'$. By induction, there exists ∇ such that $\Sigma\#\mathsf{a} : \Delta; \nabla \Longrightarrow G'$ is derivable. We can therefore derive

$$\frac{\Sigma : \mathsf{И a}.\nabla \vDash \mathsf{И a}.\nabla \quad \Sigma, X{:}\sigma : \Delta; \mathsf{И a}.\nabla, \nabla \Longrightarrow G'}{\Sigma : \Delta; \mathsf{И a}.\nabla \Longrightarrow \mathsf{И a}.G'}\ \mathsf{И}R$$

using weakening to obtain the second subderivation.

Similarly, the second part follows by induction on D, unwinding the definition of T_D in each case. We show the case for $\mathsf{И}L$.

—If $D = \mathsf{И a}{:}\nu.D'$, assume without loss of generality that $\mathsf{a} \notin \Sigma, \theta, A$. Then $\theta(D) = \mathsf{И a}.\theta(D')$ and since $T_{\mathsf{И a}.\theta(D')}(S) = \bigcup_{\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a}\ \mathsf{b})\cdot\theta(D')}(S)$, so we must have $\Sigma : \bigcup_{\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a}\ \mathsf{b})\cdot\theta(D')}(T^i_\Delta), \theta \vDash A$. By definition, this means that $\theta(A) \in \bigcup_{\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a}\ \mathsf{b})\cdot\theta(D')}(T^i_\Delta)$. Since by assumption $\mathsf{a} \notin \Sigma, \theta, A$ and $\mathsf{a} \notin \mathrm{supp}(\mathsf{И a}.D')$, we must have $\theta(A) \in T_{(\mathsf{a}\ \mathsf{a})\cdot\theta(D')}(T^i_\Delta)$. Note that $(\mathsf{a}\ \mathsf{a})\cdot\theta(D') = \theta(D')$, and $\theta : \Sigma\#\mathsf{a}$, hence $\Sigma\#\mathsf{a} : T_{\theta(D')}(T^i_\Delta), \theta \vDash A$. Consequently, by induction, there exists a ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma\#\mathsf{a} : \Delta; \nabla \xrightarrow{D'} A$. Therefore, we have

$$\frac{\Sigma : \mathsf{И a}.\nabla \vDash \mathsf{И a}.\nabla \quad \Sigma\#\mathsf{a} : \Delta; \mathsf{И a}.\nabla, \nabla \xrightarrow{D'} A}{\Sigma : \Delta; \mathsf{И a}.\nabla \xrightarrow{\mathsf{И a}.D'} A}\ \mathsf{И}L$$

Moreover, clearly $\Sigma\#\mathsf{a} : \theta \vDash \nabla$ implies $\Sigma : \theta \vDash \mathsf{И a}.\nabla$.

The complete proof can be found in Appendix B. □

Theorem 4.16 (Algebraic Completeness). *If* $\Sigma : \Delta, \theta \vDash G$ *then there exists a constraint* ∇ *such that* $\Sigma : \Delta, \theta \vDash \nabla$ *and* $\Sigma : \Delta; \nabla \Longrightarrow G$ *is derivable.*

Proof. If $\Sigma : \Delta, \theta \vDash G$, then there is some n such that $\Sigma : T^n_\Delta, \theta \vDash G$, so Proposition 4.15 applies. □

We can also extend this to a "logical" completeness result (following [Jaffar et al. 1998]), namely that if an answer C classically implies G, then there is a finite set

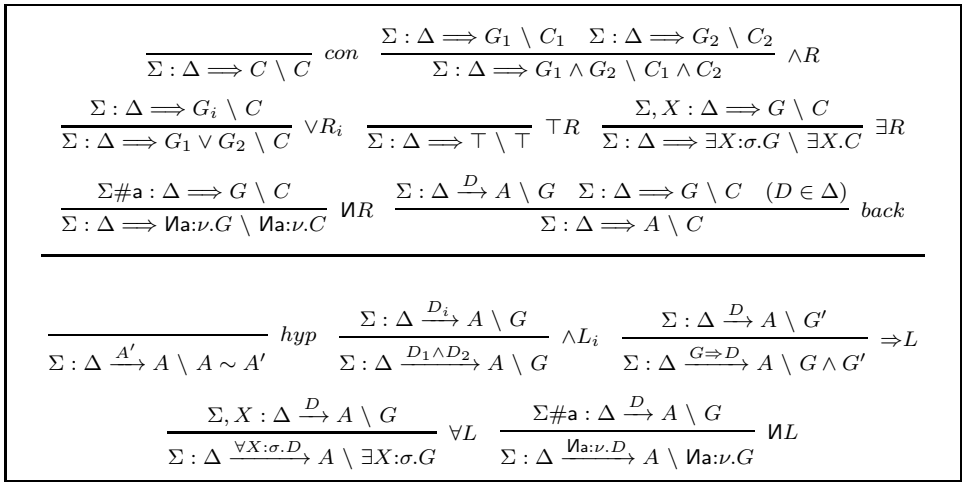

Fig. 13. Residuated uniform/focused proof search

of constraints which prove G and whose disjunction covers C. We first establish that a goal formula is classically equivalent to the disjunction (possibly infinite) of all the constraints that entail it.

LEMMA 4.17. *Let Σ be a context, Δ a program, G a goal, and $\Gamma_G = \{C \mid \Sigma : \Delta; C \Longrightarrow G\}$. Then $\Sigma : \Delta \vDash G \iff \bigvee \Gamma_G$.*

PROOF. For the forward direction, if $\Sigma : \Delta, \theta \vDash G$ then by Theorem 4.16 there exists a constraint ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma : \Delta; \nabla \Longrightarrow G$. Hence, $\bigwedge \nabla \in \Gamma_G$, so $\Sigma : \Delta, \theta \vDash \bigvee \Gamma_G$.

Conversely, if $\Sigma : \Delta, \theta \vDash \bigvee \Gamma_G$, then for some constraint $C \in \Gamma_G$, $\Sigma : \Delta, \theta \vDash C$. Consequently $\Sigma : \Delta; C \Longrightarrow G$ holds, so by Theorem 4.14, we have $\Sigma : \Delta, C \vDash G$. Since $\Sigma : \Delta, \theta \vDash C$, we conclude that $\Sigma : \Delta, \theta \vDash G$. □

THEOREM 4.18 (LOGICAL COMPLETENESS). *If $\Sigma : \Delta, C \vDash G$ then there exists a finite set of constraints Γ_0 such that $\Sigma : C \vDash \bigvee \Gamma_0$ and for each $C' \in \Gamma_0$, $\Sigma : \Delta; C' \Longrightarrow G$.*

PROOF. Again set $\Gamma_G = \{C' \mid \Sigma : \Delta; C' \Longrightarrow G\}$. By Lemma 4.17, $\Sigma : \Delta, G \vDash \bigvee \Gamma_G$. Hence, $\Sigma : \Delta, C \vDash \bigvee \Gamma_G$. By the Compactness Theorem for nominal logic [Cheney 2006a, Cor. 4.8], it follows that there is a finite subset $\Gamma_0 \subseteq \Gamma_G$ such that $\Sigma : \Delta, C \vDash \bigvee \Gamma_0$. By definition, every $C' \in \Gamma_0 \subseteq \Gamma_G$ satisfies $\Sigma : \Delta; C' \Longrightarrow G$. □

4.2.2 *The residuated system $RNL^{\Rightarrow}_{\vDash}$.* The rules in Figure 12 have the potential disadvantage that an arbitrary constraint C is allowed in the rules $\exists R, \forall L, \mathit{И}L, \mathit{И}R$. Figure 13 shows a *residuated* proof system that avoids this nondeterminism. (A similar idea is employed by Cervesato [1998]). Specifically, the judgment $\Sigma : \Delta \Longrightarrow G \setminus C$ means that given context Σ and program Δ, goal G reduces to constraint C; similarly, $\Sigma : \Delta \xrightarrow{D} A \setminus G$ means that goal formula G suffices to prove A from D.

$$
\begin{array}{lllll}
(B) & \Sigma\langle A, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\langle G, \Gamma \mid \nabla\rangle & (\text{if } \exists D \in \Delta.\Sigma : \Delta \xrightarrow{D} A \setminus G) \\
(C) & \Sigma\langle C, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\langle \Gamma \mid \nabla, C\rangle & (\nabla, C \text{ consistent}) \\
(\top) & \Sigma\langle \top, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\langle \Gamma \mid \nabla\rangle & \\
(\wedge) & \Sigma\langle G_1 \wedge G_2, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\langle G_1, G_2, \Gamma \mid \nabla\rangle & \\
(\vee_i) & \Sigma\langle G_1 \vee G_2, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\langle G_i, \Gamma \mid \nabla\rangle & \\
(\exists) & \Sigma\langle \exists X{:}\sigma.G, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma, X{:}\sigma\langle G, \Gamma \mid \nabla\rangle & \\
(И) & \Sigma\langle И\mathsf{a}{:}\nu.G, \Gamma \mid \nabla\rangle & \longrightarrow & \Sigma\#\mathsf{a}{:}\nu\langle G, \Gamma \mid \nabla\rangle &
\end{array}
$$

Fig. 14. Operational semantics transitions for nominal logic programs

Theorem 4.19 (Residuated Soundness).

(1) *If* $\Sigma : \Delta \Longrightarrow G \setminus C$ *then* $\Sigma : \Delta; C \Longrightarrow G$.

(2) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *and* $\Sigma : \Delta \xrightarrow{D} A \setminus G$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$.

Theorem 4.20 (Residuated Completeness).

(1) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *then there exists a constraint* C *such that* $\Sigma : \Delta \Longrightarrow G \setminus C$ *and* $\Sigma : \nabla \vDash C$.

(2) *If* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *then there exists goal* G *and constraint* C *such that* $\Sigma : \Delta \xrightarrow{D} A \setminus G$ *and* $\Sigma : \Delta \Longrightarrow G \setminus C$ *and* $\Sigma : \nabla \vDash C$.

Both proofs are straightforward structural inductions (see Appendix B).

4.3 Operational Semantics

We now give a CLP-style operational semantics for nominal logic programs. The rules of the operational semantics are shown in Figure 14. A program state is a triple of the form $\Sigma\langle\Gamma \mid \nabla\rangle$. Note that the backchaining step is defined in terms of residuated focused proof, $\Sigma : \Delta \xrightarrow{D} A \setminus G$.

We now state the operational soundness and completeness properties. The proofs are straightforward by cases or induction, so omitted. To simplify notation, we write $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ where $\Gamma = G_1, \ldots, G_n$ and $\vec{C} = C_1, \ldots, C_n$ to abbreviate $\Sigma : \Delta \Longrightarrow G_1 \setminus C_1, \ldots, \Sigma : \Delta \Longrightarrow G_n \setminus C_n$. In addition, we will need to reason by well-founded induction on such ensembles of derivations. We define the subderivation relation $\mathcal{D} < \mathcal{E}$ to indicate that $\mathcal{D}$ is a strict subderivation of $\mathcal{E}$, and write $\vec{\mathcal{D}} <^* \vec{\mathcal{E}}$ for the multiset ordering generated by $<$.

Proposition 4.21 amounts to showing that each operational transition corresponds to a valid manipulation on (multisets of) residuated proofs.

Proposition 4.21 (Transition Soundness). *If* $\Sigma\langle\vec{G} \mid \nabla\rangle \longrightarrow \Sigma'\langle\vec{G'} \mid \nabla'\rangle$ *and* $\Sigma' : \Delta \Longrightarrow \vec{G'} \setminus \vec{C'}$ *then there exist* $\vec{C}$ *such that*

(1) $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ *and*

(2) $\Sigma' : \nabla', \vec{C'} \vDash \nabla, \vec{C}$.

PROOF. Assume $\Sigma' : \Delta \Longrightarrow \vec{G}' \setminus \vec{C}'$ is derivable. Proof is by case decomposition on the possible transition steps. We show the case for a $\mathsf{И}$-step; the complete proof is in Appendix C. For a step ($\mathsf{И}$) of the form

$$\Sigma\langle \mathsf{И a}{:}\nu.G, \vec{G}_0 \mid \nabla\rangle \longrightarrow \Sigma\#\mathsf{a}{:}\nu\langle G, \vec{G}_0 \mid \nabla\rangle$$

Then $\Sigma' = \Sigma\#\mathsf{a}$; $\nabla' = \nabla$, $\vec{G} = \mathsf{И a}.G, \vec{G}_0$; $\vec{G}' = G, \vec{G}_0$; $\vec{C}' = C, \vec{C}_0$, so set $\vec{C} = \mathsf{И a}.C, \vec{C}_0$. For part (1), derive $\Sigma : \Delta \Longrightarrow \mathsf{И a}.G, \vec{G}_0 \setminus \mathsf{И a}.C, \vec{C}_0$ using $\mathsf{И}R$. For part (2), observe that $\Sigma\#\mathsf{a} : \nabla, C, \vec{C}_0 \vDash \mathsf{И a}.C, \vec{C}_0$. □

THEOREM 4.22 (OPERATIONAL SOUNDNESS). *if* $\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow^* \Sigma'\langle \varnothing \mid \nabla'\rangle$ *then there exists* $\vec{C}$ *such that* $\Sigma' : \nabla' \vDash \nabla, \vec{C}$ *and* $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$.

PROOF. Proof is by induction on the number of transition steps. If no steps are taken, then $\vec{G}$ is empty and $\nabla' = \nabla$, so taking $\vec{C}$ to be empty, the conclusion is trivial. Otherwise we have a step

$$\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow \Sigma_0\langle \vec{G}_0 \mid \nabla_0\rangle \longrightarrow^* \Sigma'\langle \varnothing \mid \nabla'\rangle\ .$$

By induction, there exists $\vec{C}_0$ such that $\Sigma' : \nabla' \vDash \nabla_0, \vec{C}_0$ and $\Sigma_0 : \Delta \Longrightarrow \vec{G}_0 \setminus \vec{C}_0$. Using Proposition 4.21, we can construct $\vec{C}$ such that $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ and $\Sigma_0 : \nabla_0, \vec{C}_0 \vDash \nabla, \vec{C}$. Moreover, using weakening and deduction, we can conclude that $\Sigma' : \nabla' \vDash \nabla, \vec{C}$. □

The transition completeness property (Proposition 4.23) states that for any configuration $\Sigma\langle \Gamma \mid \nabla\rangle$ such that the goals Γ have appropriate derivations in the residuated proof system, there is an operational transition step to a new state with appropriately modified derivations. This is essentially the (complicated) induction hypothesis for proving completeness of the operational semantics with respect to the other systems (Theorem 4.24).

PROPOSITION 4.23 (TRANSITION COMPLETENESS). *For any nonempty* $\vec{G}$ *and satisfiable* ∇, $\vec{C}$, *if we have derivations* $\vec{\mathcal{D}}$ *of* $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ *then for some* Σ', ∇', *and* $\vec{C}'$ *we have*

(1) $\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow \Sigma'\langle \vec{G}' \mid \nabla'\rangle$,

(2) *There exist derivations* $\vec{\mathcal{D}}'$ *of* $\Sigma' : \Delta \Longrightarrow \vec{G}' \setminus \vec{C}'$, *where* $\vec{\mathcal{D}}' <^* \vec{\mathcal{D}}$

(3) $\exists\Sigma[\nabla] \vDash \exists\Sigma'[\nabla']$

PROOF. Let $\vec{G}, \vec{C}, \nabla$ be given as above. Since $\vec{G}$ is nonempty, we must have $\vec{G} = G, \vec{G}_0$ and $\vec{C} = C, \vec{C}_0$. Proof is by case decomposition of the derivation of $\Sigma : \Delta \Longrightarrow G \setminus C$. We show the cases for $\mathsf{И}R$ and *back*; the complete proof is in Appendix C. If the derivation is of the form

$$\frac{\Sigma\#\mathsf{a} : \Delta \Longrightarrow G \setminus C}{\Sigma : \Delta \Longrightarrow \mathsf{И a}.G \setminus \mathsf{И a}.C}\ \mathsf{И}R$$

$\vec{G} = \mathsf{И a}.G, \vec{G}_0$ and $\vec{C} = \mathsf{И a}.C, \vec{C}_0$. Setting $\Sigma' = \Sigma\#\mathsf{a}; \nabla' = \nabla; \vec{G}' = G, \vec{G}_0; \vec{C}' = C, \vec{C}_0$; we can take the operational step $\Sigma\langle \mathsf{И a}.G, \vec{G}_0 \mid \nabla\rangle \longrightarrow \Sigma\#\mathsf{a}\langle G, \vec{G}_0 \mid \nabla\rangle$. In addition, for (2) we can obtain smaller subderivations of $\Sigma\#\mathsf{a} : \Delta \Longrightarrow G, \vec{G}_0 \setminus C, \vec{C}_0$

from the given derivations, and for (3) observe that $\exists\Sigma[\nabla, И\mathsf{a}.C, \vec{C}_0] \vDash \exists\Sigma\#\mathsf{a}[\nabla, C, \vec{C}_0]$ since a is not free in $\nabla, \vec{C}_0$.

For a derivation of the form

$$\frac{\Sigma : \Delta \xrightarrow{D} A \setminus G' \quad \Sigma : \Delta \Longrightarrow G' \setminus C \quad (D \in \Delta)}{\Sigma : \Delta \Longrightarrow A \setminus C}\ \mathit{back}$$

we have $\vec{G} = A, \vec{G}_0$ and $\vec{C} = C, \vec{C}_0$. Set $\Sigma = \Sigma'$; $\vec{G}' = G', \vec{G}_0$; $\vec{C}' = C, \vec{C}_0$; $\nabla' = \nabla$. Using the first subderivation, we can take a backchaining step $\Sigma\langle A, \vec{G}_0 \mid \nabla\rangle \longrightarrow \Sigma\langle G', \vec{G}_0 \mid \nabla\rangle$. Moreover, for part (2), using the second subderivation we obtain a smaller derivation $\Sigma : \Delta \Longrightarrow G', \vec{G}_0 \setminus C, \vec{C}_0$, and part (3) is trivial. □

Theorem 4.24 (Operational Completeness). *If $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ and $\nabla, \vec{C}$ is satisfiable then for some Σ' and ∇', we have $\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow^* \Sigma'\langle \varnothing \mid \nabla'\rangle$ and $\exists\Sigma[\nabla, \vec{C}] \vDash \exists\Sigma'[\nabla']$.*

Proof. The proof is by induction on the length of $\vec{G}$ and the sizes of the derivations $\vec{\mathcal{D}}$ of $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$. If $\vec{G}$ is empty, then we are done. Otherwise, using Proposition 4.23, there exist Σ_0, $\vec{G}_0$, $\vec{C}_0$, and ∇_0, such that

$$\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow \Sigma_0\langle \vec{G}_0 \mid \nabla_0\rangle \quad \Sigma_0 : \Delta \overset{\vec{\mathcal{D}}'}{\Longrightarrow} \vec{G}_0 \setminus \vec{C}_0 \quad \exists\Sigma[\nabla, \vec{C}] \vDash \exists\Sigma_0[\nabla_0, \vec{C}_0]$$

The derivations $\vec{\mathcal{D}}'$ are smaller than $\vec{\mathcal{D}}$, and the satisfiability of $\nabla, \vec{C}$ implies that $\nabla_0, \vec{C}_0$ is also satisfiable, so the induction hypothesis applies. Accordingly, construct Σ', ∇' such that

$$\Sigma_0\langle \vec{G}_0 \mid \nabla_0\rangle \longrightarrow^* \Sigma'\langle \varnothing \mid \nabla'\rangle \quad \exists\Sigma[\nabla_0, \vec{C}_0] \vDash \exists\Sigma'[\nabla']$$

Chaining the transitions and entailments, we conclude

$$\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow \Sigma_0\langle \vec{G}_0 \mid \nabla_0\rangle \longrightarrow^* \Sigma'\langle \varnothing \mid \nabla'\rangle \quad \exists\Sigma[\nabla, \vec{C}] \vDash \exists\Sigma_0[\nabla_0, \vec{C}_0] \vDash \exists\Sigma'[\nabla']$$

as desired. □

4.4 Summary

The goal of this section has been to present and show the equivalence of model-theoretic, proof-theoretic, and operational presentations of the semantics of nominal logic programs. We abbreviate $\Sigma\langle G \mid \varnothing\rangle \longrightarrow^* \Sigma; \Sigma'\langle \varnothing \mid C\rangle$ as $\Sigma\langle G\rangle \Downarrow \exists\Sigma'[C]$. The soundness and completeness theorems we have established can be chained together as follows to summarize these results:

Corollary 4.25. *If $\Sigma\langle G\rangle \Downarrow \nabla$ then:*

(1) there exists C such that $\Sigma : \nabla \vDash C$ and $\Sigma : \Delta \Longrightarrow G \setminus C$;
(2) $\Sigma : \Delta; \nabla \Longrightarrow G$; and
(3) $\Sigma : \Delta, \nabla \vDash G$

Proof. Immediate using Theorem 4.22, Theorem 4.19, and Theorem 4.14. □

Corollary 4.26.

(1) *If* $\Sigma : \Delta \Longrightarrow G \setminus C$ *and* C *is satisfiable then for some* ∇*, we have* $\Sigma\langle G\rangle \Downarrow \nabla$ *and* $\Sigma : C \vDash \nabla$.

(2) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *and* ∇ *is satisfiable then for some* ∇'*, we have* $\Sigma\langle G\rangle \Downarrow \nabla'$ *and* $\Sigma : \nabla \vDash \nabla'$.

(3) *If* $\Sigma : \Delta, \theta \vDash G$ *then for some* ∇*, we have* $\Sigma\langle G\rangle \Downarrow \nabla$ *and* $\Sigma : \theta \vDash \nabla$.

(4) *If* $\Sigma : \Delta, C \vDash G$ *then there exists a finite collection of constraints* $\vec{\nabla}$ *such that* $\Sigma\langle G\rangle \Downarrow \nabla_i$ *for each* $\nabla_i \in \vec{\nabla}$ *and* $\Sigma : C \vDash \nabla_1 \vee \cdots \vee \nabla_n$.

Proof. Immediate using Theorem 4.24, Theorem 4.20, Theorem 4.16, Theorem 4.18. □

These results ensure that the operational semantics computes all (and only) correct solutions with respect to nominal logic, so the proof-theoretic and model-theoretic semantics can be used to reason about the behavior of programs; this is often much easier than reasoning about the operational semantics, as we shall now demonstrate.

5. APPLICATIONS

5.1 Correctness of elaboration

In an implementation, program clauses are often *elaborated* into a normal form $\forall\Sigma[G \Rightarrow A]$ which is easier to manipulate and optimize. We define the elaboration of a program clause or program as the result of normalizing it with respect to the following rewrite system:

$$
\begin{array}{rcl}
G \Rightarrow \top & \rightsquigarrow & \top \\
D \wedge \top & \rightsquigarrow & D \\
\top \wedge D & \rightsquigarrow & D \\
\forall X.\top & \rightsquigarrow & \top \\
\text{И}\mathsf{a}.\top & \rightsquigarrow & \top \\
\Delta, D \wedge D' & \rightsquigarrow & \Delta, D, D' \\
\Delta, \top & \rightsquigarrow & \Delta
\end{array}
\qquad
\begin{array}{rcll}
G \Rightarrow G' \Rightarrow D & \rightsquigarrow & G \wedge G' \Rightarrow D & \\
G \Rightarrow D \wedge D' & \rightsquigarrow & (G \Rightarrow D) \wedge (G \Rightarrow D') & \\
G \Rightarrow \forall X.D & \rightsquigarrow & \forall X.(G \Rightarrow D) & (X \notin FV(G)) \\
G \Rightarrow \text{И}\mathsf{a}.D & \rightsquigarrow & \text{И}\mathsf{a}.(G \Rightarrow D) & (\mathsf{a} \notin \mathit{supp}(G)) \\
\forall X.(D \wedge D') & \rightsquigarrow & \forall X.D \wedge \forall X.D' & \\
\text{И}\mathsf{a}.(D \wedge D') & \rightsquigarrow & \text{И}\mathsf{a}.D \wedge \text{И}\mathsf{a}.D' &
\end{array}
$$

It is straightforward to show that this system is terminating and confluent (up to α- and multiset-equality) and that elaborated programs consist only of closed formulas of the form $\forall\Sigma[G \Rightarrow A]$. Moreover, this translation preserves the meaning of the program:

Theorem 5.1 Correctness of elaboration.

(1) *If* $\Delta \rightsquigarrow \Delta'$ *then* $\Sigma : \Delta; \nabla \Longrightarrow G$ *iff* $\Sigma : \Delta'; \nabla \Longrightarrow G$.

(2) *If* $\Delta \rightsquigarrow \Delta'$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *iff* $\Sigma : \Delta'; \nabla \xrightarrow{D} A$.

(3) *If* $D \rightsquigarrow D'$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *iff* $\Sigma : \Delta; \nabla \xrightarrow{D'} A$.

Proof. Each part is a straightforward induction on derivations and case decomposition on the possible rewriting steps. We show a few representative cases. All omitted cases are in the proof in Appendix D.

For part (1), proof is by induction on the given derivation. In this case, we need to consider the possible rewrite step taken on Δ. Writing D for the selected formula $D \in \Delta$, there are four possibilities:

—The rewrite step does not affect D. Hence, $D \in \Delta'$. Then we have

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D} A \quad (D \in \Delta')}{\Sigma : \Delta'; \nabla \Longrightarrow A}\ sel$$

—The rewrite step eliminates $D = \top$ from Δ. This case is vacuous because there can be no derivation with focused formula $\top$.

—The rewrite step splits $D = D_1 \wedge D_2 \in \Delta$; thus, $\Delta = \Delta_0, D_1 \wedge D_2$ and $\Delta' = \Delta, D_1, D_2$. Then we have

$$\frac{\dfrac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \qquad D_1 \wedge D_2 \in \Delta}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D_i} A \quad D_i \in \Delta'}{\Sigma : \Delta'; \nabla \Longrightarrow A}\ sel$$

—The rewrite step rewrites $D \rightsquigarrow D'$; thus, $D' \in \Delta'$, and using parts (2) and (3) we can obtain

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D'} A \quad (D' \in \Delta')}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

For part (2), all of the cases are straightforward.

For part (3), proof is by induction on the structure of derivations and of the possible rewriting steps. There are several easy cases in which the rewriting step takes place "deep" in the term. The remaining cases involve a rewriting step at the root of D. Of these, we show only the only novel cases involving И.

If the rewriting step is $G \Rightarrow \text{И}\mathsf{a}.D \rightsquigarrow \text{И}\mathsf{a}.(G \Rightarrow D)$, where $\mathsf{a} \notin \mathrm{supp}(G, \Sigma)$, then we can derive

$$\frac{\Sigma : \Delta; \nabla \overset{\mathcal{D}_1}{\Longrightarrow} G \qquad \dfrac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}.D} A}\ \text{И}L \overset{\mathcal{D}_2}{}}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow \text{И}\mathsf{a}.D} A}\ \Rightarrow L$$

if and only if we can also derive

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \qquad \dfrac{\Sigma\#\mathsf{a} : \Delta; \nabla, C \overset{\mathcal{D}_1'}{\Longrightarrow} G \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A \overset{\mathcal{D}_2}{}}{\Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{G \Rightarrow D} A}\ \Rightarrow L}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}.(G \Rightarrow D)} A}\ \text{И}L$$

since a is not mentioned in G or Σ.

If the rewriting step is $\text{И}\mathsf{a}.(D_1 \wedge D_2) \rightsquigarrow \text{И}\mathsf{a}.D_1 \wedge \text{И}\mathsf{a}.D_2$ then for $i \in \{1, 2\}$ we can derive

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \dfrac{\Sigma\#\mathsf{a} : \Delta; \nabla, C \overset{\mathcal{D}}{\xrightarrow{D_i}} A}{\Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}.(D_1 \wedge D_2)} A}\ \text{И}L$$

if and only if we can derive

$$\dfrac{\dfrac{\mathcal{D}}{\Sigma : \nabla \vDash \mathit{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D_i} A}\ \mathit{И}L}{\Sigma : \Delta; \nabla, C \xrightarrow{\mathit{И}\mathsf{a}.D_i} A}\ \wedge L_i$$
$$\Sigma : \Delta; \nabla \xrightarrow{\mathit{И}\mathsf{a}.D_1 \wedge \mathit{И}\mathsf{a}.D_2} A$$

This completes the proof. □

5.2 Completeness of ≈-resolution

In general, full nominal constraint solving is intractable, and the only known algorithm is somewhat involved [Cheney 2004a; 2005a]. However, Urban, Pitts, and Gabbay's nominal unification algorithm solves a tractable special case, specifically, it works for constraints involving only ≈ or # that are satisfy the following *ground name restriction*:

Definition 5.2. We say that a term, formula, or constraint is *(ground) name-restricted* if, for every subformula or subterm of one of the forms

$$a \# t \quad (a\ b) \cdot t \quad \langle a \rangle t$$

the subterms a, b are ground names.

We refer to Urban, Pitts and Gabbay's nominal unification algorithm as *restricted nominal unification*. It is an attractive idea to limit nominal logic programs to the name-restricted fragment and use restricted nominal unification algorithm for resolution. In the proof theoretic semantics, we can model this behavior by replacing the *hyp* rule with $hyp_{\approx}$

$$\dfrac{\Sigma : \nabla \vDash A \approx A'}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp_{\approx}$$

in which the stronger condition $\Sigma : \nabla \vDash A \approx A'$ is required to conclude $\Sigma : \Delta; \nabla \xrightarrow{A'} A$. We write $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G$ and $\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A$ for uniform or focused proofs in which $hyp_{\approx}$ is used instead of *hyp*, and refer to such proofs as equational resolution (or ≈*-resolution*) proofs. It is easy to verify that ≈-resolution proofs are sound with respect to ordinary derivations and that all constraints arising in such proofs are name-restricted.

Unfortunately, ≈-resolution is incomplete relative to the full system, because unlike in first-order logic, two atomic formulas can be logically equivalent, but not equal as nominal terms. Equational resolution fails to find solutions that depend on this aspect of nominal logic. The simplest example is the single program clause

$$\mathit{И}\mathsf{a}.p(\mathsf{a}).$$

If we try to solve the goal $p(X)$ against this program, then we get a solution $X = \mathsf{a}'$ for some fresh name a'; up to equivariance, this is the most general solution. However, if pose the query $p(\mathsf{a})$ then proof search fails, yet logically, $\exists X.p(X) \iff \mathit{И}\mathsf{a}.p(\mathsf{a})$.

This example shows that equational resolution is incomplete for name-restricted programs. Moreover, ≈-resolution over name-restricted terms is **NP**-complete

via an easy reduction from the **NP**-completeness of restricted equivariant unification [Cheney 2004a]. Thus, there is little hope that a polynomial-time resolution algorithm for name-restricted nominal logic programs can be found.

A natural next question is whether further syntactic restrictions beyond name-restriction can guarantee completeness for equational resolution, yet still permit writing interesting nominal logic programs. Such a criterion does exist. However, before presenting it and proving its correctness, we describe some plausible-seeming, but insufficient attempts.

Example 5.3. The clause $И\mathsf{a}.p(\mathsf{a})$ mentions an "unbound" name a in its head. However, if we impose the natural-seeming further restriction that names may only appear bound in the head of the clause, there are still clauses and goals for which proof search is incomplete. For example,

$$И\mathsf{a}.\forall X.q(\langle\mathsf{a}\rangle X, X).$$

logically implies the goal formula $q(\langle\mathsf{a}\rangle\mathsf{a}, \mathsf{b})$, but proof search for this goal fails.

If we forbid names *anywhere* in the head of the clause, this is also not enough, as the following program illustrates:

$$И\mathsf{a}.\forall X.r(X) :- X \approx \mathsf{a}.$$

for this program logically implies $И\mathsf{a}.r(\mathsf{a})$ but this answer cannot be found by $\approx$-resolution.

On the other hand, forbidding all names anywhere in a clause seems to be enough to guarantee completeness, but this means that only "first-order" Horn clauses not mentioning names, abstraction, freshness, or swapping can be used as program clauses. While this does mean that ordinary first-order logic programs can be executed efficiently over nominal terms, it rules out all interesting nominal logic programs.

Now we consider criteria which do ensure that $\approx$-resolution is complete yet permit interesting nominal logic programs. One interesting example identified by Urban and Cheney [2005]. The key idea is that names in the head of the clause are all right as long as they are inessential to the meaning of the clause. Specifically, if a name a appears in a clause, then it must be fresh for all of the terms appearing in the head of the clause. However, this condition turns out to be somewhat difficult to analyze.

We say that a program clause is И-goal if it has no subformula of the form $И\mathsf{a}.D$. However, И-quantified goals $И\mathsf{a}.G$ are allowed. Such goals and program clauses are generated by the BNF grammar:

$$\begin{aligned} G &::= \top \mid A \mid C \mid G \wedge G' \mid G \vee G' \mid \exists X.G \mid И\mathsf{a}.G \\ D &::= \top \mid A \mid D \wedge D' \mid G \Rightarrow D \mid \forall X.D \end{aligned}$$

Example 5.4. Although the *tc* program of Section 2 is not И-goal, its third clause is equivalent to the И-goal formula

$$tc(Ctx, lam(F), fn(T,U)) :- И\mathsf{x}.F \approx \langle\mathsf{x}\rangle E, tc([(\mathsf{x},T)|Ctx], E, U).$$

Technically, the above clause imposes the additional restriction that $\mathsf{x} \# T, U$, but simple types do not contain variable names, so this does not affect the behavior of

the program.

As a first step towards proving that $\approx$-backchaining is complete for $\forall\nabla$-goal programs, we show that $\approx$-backchaining derivations are stable under application of permutations for such programs.

LEMMA 5.5. *Let Δ be a $\forall\nabla$-goal program and π be a type-preserving permutation of names in Σ.*

(1) If $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G$ then $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}G$.

(2) If $\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A$ then $\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot D}_{\approx} \pi{\cdot}A$.

PROOF. By induction on derivations. For part (1), most cases are straightforward; we show representative cases $\exists R$, $\forall\nabla R$, and *sel*.

—For case $\exists R$, we have

$$\frac{\Sigma : \nabla \vDash \exists X.C[X] \quad \Sigma, X : \Delta; \nabla, \overset{\mathcal{D}}{C[X]} \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \exists X.G}\ \exists R$$

Note that $\pi{\cdot}\exists X.G[X] = \exists X.\pi{\cdot}G[\pi^{-1}{\cdot}X]$. By induction,

$$\Sigma, X : \Delta; \nabla, \overset{\mathcal{D}}{C[X]} \Longrightarrow_{\approx} G \longmapsto \Sigma, X : \Delta; \nabla, \overset{\mathcal{D}'}{C[X]} \Longrightarrow_{\approx} \pi{\cdot}G[X]\ .$$

Since π is invertible, we can substitute $Y = \pi{\cdot}X$ to obtain $\mathcal{D}'' :: \Sigma, Y : \Delta; \nabla, C[\pi^{-1}{\cdot}Y] \Longrightarrow_{\approx} \pi{\cdot}G[\pi^{-1}{\cdot}Y]$; moreover, clearly, $\Sigma : \nabla \vDash \exists Y.C[\pi^{-1}{\cdot}Y]$, so we can conclude

$$\frac{\Sigma : \nabla \vDash \exists Y.C[\pi^{-1}{\cdot}Y] \quad \Sigma, Y : \Delta; \nabla, C[\pi^{-1}{\cdot}Y] \overset{\mathcal{D}''}{\Longrightarrow}_{\approx} \pi{\cdot}G[\pi^{-1}{\cdot}Y]}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}\exists X.G}\ \exists R\ .$$

—For case $\forall\nabla R$, we have derivation

$$\frac{\Sigma : \nabla \vDash \forall\nabla\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \overset{\mathcal{D}}{\nabla}, C \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \forall\nabla\mathsf{a}{:}\nu.G}\ \forall\nabla R \longmapsto \frac{\Sigma : \nabla \vDash \forall\nabla\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \overset{\mathcal{D}'}{\nabla}, C \Longrightarrow_{\approx} \pi{\cdot}G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}(\forall\nabla\mathsf{a}{:}\nu.G)}\ \forall\nabla R$$

since $\pi{\cdot}\forall\nabla\mathsf{a}{:}\nu.G = \forall\nabla\mathsf{a}{:}\nu.\pi{\cdot}G$, (since, without loss, $\mathsf{a} \notin FN(\Sigma) \cup \mathrm{supp}(\pi)$). The derivation $\mathcal{D}' :: \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow_{\approx} \pi{\cdot}G$ is obtained by induction.

—For case *sel*,

$$\frac{\overset{\mathcal{D}}{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A} \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} A}\ sel \longmapsto \frac{\overset{\mathcal{D}'}{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} \pi{\cdot}A} \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}A}\ sel$$

using induction hypothesis (2) to derive $\mathcal{D}'$ from $\mathcal{D}$, and the fact that $\pi{\cdot}D = D$ (because $D \in \Delta$ is closed).

For part (2), all cases are straightforward; we show *hyp* and $\Rightarrow L$. Case $\forall L$ requires a change of variables argument similar to $\exists R$. Case $\forall\nabla L$ is vacuous.

—Case *hyp*

$$\frac{\Sigma : \nabla \vDash A' \approx A}{\Sigma : \Delta; \nabla \xrightarrow{A'}_{\approx} A}\ hyp \longmapsto \frac{\Sigma : \nabla \vDash \pi{\cdot}A' \approx \pi{\cdot}A}{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot A'}_{\approx} \pi{\cdot}A}\ hyp$$

since $\Sigma : A' \approx A \vDash \pi{\cdot}A' \approx \pi{\cdot}A$.

—Case $\Rightarrow L$

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D}_{\approx} A} \Rightarrow L \longmapsto \frac{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} \pi{\cdot}A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}G}{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot (G \Rightarrow D)}_{\approx} \pi{\cdot}A} \Rightarrow L$$

where the subderivations are obtained by induction; this suffices because $\pi{\cdot}(G \Rightarrow D) = \pi{\cdot}G \Rightarrow \pi{\cdot}D$.

The complete proof is given in Appendix E. □

We can now show that $\approx$-derivations are complete for И-goal programs.

THEOREM 5.6. *If Δ is И-goal then*

(1) *If $\Sigma : \Delta; \nabla \Longrightarrow G$ is derivable, then $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G$ is derivable.*

(2) *If $\Sigma : \Delta; \nabla \xrightarrow{D} A$ is derivable, there exists a π such that $\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} A$ is derivable.*

PROOF. The proof is by induction on derivations. For part (1), the most interesting case is *sel*; the rest are straightforward. For a derivation ending in *sel*, we have

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

for some closed $D \in \Delta$. By induction hypothesis (2), for some π, $\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} A$ holds. However, since D is closed, $\pi{\cdot}D = D \in \Delta$ so we may conclude

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} A}\ sel$$

For part (2), the interesting cases are *hyp* and ИL; $\forall L$ requires a change of variables (as in Lemma 5.5).

—For *hyp*, we have

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp$$

By definition $\Sigma : \nabla \vDash A' \sim A$ means there exists a π such that $\Sigma : \nabla \vDash \pi{\cdot}A' \approx A$, so

$$\frac{\Sigma : \nabla \vDash \pi{\cdot}A' \approx A}{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot A'}_{\approx} A}\ hyp$$

—Case $\Rightarrow L$: Using both induction hypotheses, and then Lemma 5.5(1), we can transform the derivation as follows:

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A} \Rightarrow L \longmapsto \frac{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}G}{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot G \Rightarrow \pi \cdot D}_{\approx} A} \Rightarrow L$$

—Case ИL is vacuous.

The complete proof is given in Appendix E. □

Note that Theorem 5.6 fails if $И L$ is allowed: for example, faced with a derivation

$$\frac{\Sigma\#\mathsf{a} : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}.D} A}\ И L$$

we can obtain $\Sigma\#\mathsf{a} : \Delta; \nabla \xrightarrow{\pi\cdot D}_{\approx} A$ by induction, but since π may mention a, it is not possible in general to conclude $\Sigma : \Delta; \nabla \xrightarrow{\pi'\cdot И\mathsf{a}.D}_{\approx} \pi'\cdot A$ for some π'. (This can be seen for $D = p(\mathsf{a}), A = p(\mathsf{b}), \pi = (\mathsf{a}\ \mathsf{b})$ in Example 5.3.)

6. COMPARISON WITH PREVIOUS WORK

Several techniques for providing better handling of syntax with bound names in logic programming settings have been considered:

—Higher-order logic programming and higher-order abstract syntax [Nadathur and Miller 1998; Pfenning and Elliott 1989; Pfenning 1991; Pfenning and Schürmann 1999]

—Lambda-term abstract syntax, a variation on higher-order abstract syntax based on Miller's *higher-order patterns* [Miller 1991]

—Qu-Prolog, a first-order logic programming language with binding and substitution constraints [Staples et al. 1996; Cheng et al. 1991; Nickolas and Robinson 1996; Clark et al. 2001]

—Logic programming based on *binding algebras*, an approach to the semantics of bound names based on functor categories [Hamana 2001; Fiore et al. 1999; Hofmann 1999].

We also relate our approach with functional programming languages that provide built-in features for name-binding, such as ML_λ [Miller 1990], FreshML [Pitts and Gabbay 2000; Shinwell et al. 2003; Shinwell and Pitts 2005], and Delphin [Schürmann et al. 2005], as well as recent efforts to provide nominal abstract syntax as a lightweight language extension [Pottier 2005; Cheney 2005c].

6.1 Logic programming with names and binding

6.1.1 *Higher-order logic programming.* Higher-order abstract syntax [Pfenning and Elliott 1989] is a powerful and elegant approach to programming with names and binding that is well-supported by higher-order logic programming languages such as λProlog [Nadathur and Miller 1998] or Twelf [Pfenning 1991; Pfenning and Schürmann 1999]. In higher-order logic programming, we consider logic programs to be formulas of a higher-order logic such as Church's simple type theory Church [1940] or the logical framework LF [Harper et al. 1993]. Higher-order logic programming provides logically well-founded techniques for modularity and abstraction [Miller 1989] and provides advanced capabilities for programming with abstract syntax involving bound names, and for programming relations with "local" hypotheses.

These capabilities are ideal for programming a wide variety of type systems, program transformations, and theorem provers [Hannan and Miller 1988; Pfenning

1991; Felty 1993; Nadathur and Miller 1998]. Thus higher-order logic programming is an excellent tool for prototyping and designing type systems and program transformations.

While this approach is elegant and powerful, it has some disadvantages as well. These disadvantages seem inextricably connected to higher-order abstract syntax's main advantage: the use of constants of higher-order type to describe object language binding syntax, meta-language variables to encode object variables, and meta-language hypotheses and contexts to encode object-language assumptions and contexts. In particular, the fact that object-language names "disappear" into meta-level variables means that computations that involve comparing names (such as *cconv*) or generating fresh names are difficult to perform in a higher-order encoding.

Another drawback of the higher-order approach is that "elegant" encodings work well only when the inherent properties of the meta-language concepts are shared by the object language. In particular, if the LF context is used for the context(s) of the object language, then the latter inherits the properties of the former, such as weakening and contraction. Many interesting systems have unusual contexts or binding behavior, especially substructural type systems [Girard 1987; O'Hearn and Pym 1999], logics of imperative programs [Mason 1987; Harel et al. 2000; Reynolds 2002], and low-level syntax transformations such as closure conversion. Representations of these languages in pure higher-order logic (or LF) seem disproportionately difficult to program and reason about. Of course, such programs can still be written as higher-order logic programs, only without making full use of higher-order abstract syntax. This can result in nondeclarative (and nonintuitive) programs and complicates reasoning about the object system.

One remedy is to extend the meta-language with new features that make it possible to encode larger classes of object languages elegantly. This approach has been employed primarily in the setting of LF; examples include linearity (Linear LF [Cervesato and Pfenning 2002]) and monadic encapsulation of effects (Concurrent LF [Watkins et al. 2003]). Nevertheless, there remain many logics and programming languages which are difficult to encode elegantly in any extant LF-style system.

6.1.2 *Logic programming with higher-order patterns.* L_λ is a restricted form of higher-order logic programming introduced by Miller [1991]. In L_λ, occurrences of meta-variables in unification problems are required to obey the *higher-order pattern constraint*: namely, each such meta-variable may only occur as the head of an application to a sequence of distinct *bound* variables. For example, $\lambda x.F\ x$ is a pattern but $\lambda x.F\ x\ x$ and $\lambda x.x(FX)$ are not. The higher-order pattern restriction guarantees that most general unifiers exist, and that unification is decidable.

However, built-in capture-avoiding substitution for arbitrary terms is not available in L_λ. In full λProlog, the beta-reduction predicate can be encoded as

```
beta (app (lam (x\M x)) N) (M N).
```

but this is not a higher-order pattern because of the subterm `M N`. Instead, substitution must be programmed explicitly in L_λ, though this is not difficult:

```
beta (app (lam (x\E x)) E') E'' :- subst (x\E x) E' E''.
```

```
subst (x\x) E E.
subst (x\app (E1 x) (E2 x)) E (app E1' E2')
  :- subst E1 E E1', subst E2 E E2'.
subst (x\lam y\E1 x y) E (lam y\E1' y)
  :- pi y\ (subst (x\y) E y -> subst (x\E1 x y) E (E1' y)).
```

This definition involves only higher-order patterns. In L_λ, the only substitutions permitted are of the form Miller calls β_0:

$$(\lambda x.M)\ y = M[y/x]$$

that is, in which a bound variable is replaced with another bound variable.

There are several interesting parallels between L_λ and αProlog (and nominal unification and L_λ unification [Urban et al. 2004]). The name-restricted fragment of nominal logic programming which underlies the current αProlog implementation seems closely related to L_λ. It seems possible to translate many programs directly from one formalism to the other, for example, by replacing local hypotheses with an explicit context. The proof-theoretic semantics in this paper may be useful for further investigating this relationship.

Miller and Tiu have investigated logics called $FO\lambda^{\Delta\nabla}$ and LG^ω which include a novel quantifier ∇ that quantifies over "generic" objects [Miller and Tiu 2003; Tiu 2006]. The ∇-quantifier has many properties in common with N; Miller and Tiu argue that ∇ provides the right logical behavior to encode "fresh name" constraints such as arise in encoding (bi)similarity in the π-calculus. However, $FO\lambda^{\Delta\nabla}$ has primarily been employed as a foundation for encoding and reasoning about languages, not as the basis of a logic programming language per se. Moreover, in Miller and Tiu's approach, higher-order abstract syntax is used to deal with binding and substitution at an implicit level, whereas in our approach, names, binding, and freshness are explicitly axiomatized.

6.1.3 *Qu-Prolog.* Qu-Prolog [Staples et al. 1996; Cheng et al. 1991; Nickolas and Robinson 1996] is a logic programming language with built-in support for object languages with variables, binding, and capture-avoiding substitution. It extends Prolog's (untyped) term language with constant symbols denoting object-level variables and a built-in simultaneous capture-avoiding substitution operation $t\{t_1/x_1, \ldots, t_n/x_n\}$. Also, a binary predicate $x\ not\_free\_in\ t$ is used to assert that an object-variable x does not appear in a term t. Certain identifiers can be declared as binders or *quantifiers*; for example, *lambda* could be so declared, in which case the term *lambda x t* is interpreted as binding x in t. Unlike in higher-order abstract syntax, quantifier symbols are not necessarily λ-abstractions, so Qu-Prolog is not simply a limited form of higher-order logic programming. Qu-Prolog does not provide direct support for name-generation; instead name-generation is dealt with by the implementation during execution as in higher-order abstract syntax.

Qu-Prolog is based on a classical theory of names and binding described in terms of substitution. Like higher-order unification, Qu-Prolog's unification problem is undecidable, but in practice a semidecision procedure based on delaying "hard" subproblems seems to work well [Nickolas and Robinson 1996].

Qu-Prolog enjoys a mature implementation including a compiler for Qu-Prolog written in Qu-Prolog. Many interesting programs have be written in Qu-Prolog, in-

cluding interactive theorem provers, client/server and database applications [Clark et al. 2001]. Relations such as λ-term typability can be programmed essentially the same as in αProlog. As with higher-order abstract syntax, Qu-Prolog's built-in substitution operation is extremely convenient.

Formal investigations of Qu-Prolog have been limited to the operational semantics and unification algorithm. There is no denotational or proof-theoretic semantics explaining the behavior of names and binding in Qu-Prolog. Qu-Prolog is untyped and there is no distinction between names and ordinary Prolog constants. There is no analogue of the $\forall$-quantifier or the equivariance or freshness principles. It may be possible to define a clearer denotational semantics for Qu-Prolog programs in terms of nominal logic. This could be useful for relating the expressiveness of αProlog and Qu-Prolog. Conversely, it may be interesting to add a Qu-Prolog-like built-in substitution operation (and associated unification techniques) to αProlog.

6.1.4 *Logic programming with binding algebras.* Fiore et al. [1999] and Hofmann [1999] introduced *binding algebras* and techniques for reasoning about abstract syntax with binding using functor categories. Hamana [2001] developed a unification algorithm and logic programming language for programming with binding algebra terms involving name-abstraction $[a]t$, name-application $t@a$, name occurrences $var(a)$, injective renamings $\xi = [x := y, x_2 := y_2, \ldots]$, and first-order function symbols and constants.

Hamana's unification algorithm unifies up to β_0-equivalence of bound names with respect to name-application. Hamana employs a type system that assigns each term a type and a set of names that may appear free in the term. Hamana's unification algorithm appears to generalize higher-order pattern unification; since names in application sequences do not have to be distinct, however, most general unifiers do not exist; for example $[x]F \mathbin{@} x \mathbin{@} x \approx^? [y]G \mathbin{@} y$ has two unifiers, $F = [x][y]y$ and $F = [x][y]x$.

Many of the example programs of Section 2 can also be programmed using Hamana's programming language. For example, capture-avoiding substitution is given as an example by Hamana [2001]. However, because binding algebras are based on arbitrary renamings, rather than injective renamings, it may be difficult to write programs such as *cconv* that rely on distinguishing names. In addition, since the names free in a term must appear in the term's type, some programs may require more involved type annotations or may be ruled out by the type system.

6.2 Functional programming with names and binding

6.2.1 ML_λ. Miller also proposed a functional language extending Standard ML to include an *intensional function type* $\tau \Rightarrow \tau'$ populated by "functions that can be analyzed at run-time", that is, higher-order patterns [Miller 1990]. This language is called ML_λ and supports functional programming with λ-term abstract syntax using the intensional function type. Since higher-order pattern unification and matching are decidable, programs in ML_λ can examine the structure of intensional function values, in contrast to ordinary function values which cannot be examined, only applied to data. Miller's original proposal left many issues open for future consideration; Pasalic et al. [2000] developed an operational semantics and prototype implementation of a language called DALI, which was inspired by ML_λ.

6.2.2 *FreshML.* FreshML [Pitts and Gabbay 2000; Shinwell et al. 2003; Shinwell and Pitts 2005] is a variant of ML (or Objective Caml) that provides built-in primitives for names and binding based on nominal abstract syntax. FreshML was an important source of inspiration for αProlog. At present FreshML and αProlog provide similar facilities for dealing with nominal abstract syntax. Arguably, because of the similarities between higher-order patterns and nominal terms [Urban et al. 2004; Cheney 2005b], FreshML can be viewed as an alternative realization of ML_λ.

The main differences are

—FreshML's treatment of name-generation uses side-effects, whereas αProlog uses nondeterminism.

—There are no ground names in FreshML programs; instead, names are always manipulated via variables.

—FreshML currently provides more advanced forms of name-binding (such as binding a list of names simultaneously).

—FreshML provides richer higher-order programming features.

Conversely, there are many programs that can be written cleanly in αProlog's logical paradigm but not so cleanly in FreshML's functional paradigm, such as typechecking relations and nondeterministic transition systems.

6.2.3 *Delphin.* Another language which draws upon ML_λ is Delphin. Delphin is a functional language for programming with higher-order abstract syntax and dependent types [Schürmann et al. 2005]. Because ordinary recursion principles do not work for higher-order encodings that violate the positivity restriction, Delphin provides novel features (based on earlier work in the context of Twelf [Schürmann 2001b; 2001a]) for writing such programs. This approach seems very powerful, but also potentially more complex than nominal techniques. For example, Delphin programs may be nondeterministic and produce non-ground answers, because the underlying higher-order matching problems needed for pattern matching may lack most general unifiers. At present a prototype called Elphin has been implemented.

6.2.4 *Cαml.* Pottier [2005] has developed a tool for OCaml called Cαml. Cαml translates high-level, OCaml-like specifications of the binding structure of a language to ordinary OCaml type declarations and code for performing pattern matching and fold-like traversals of syntax trees. Cαml uses a swapping-based nominal abstract syntax technique internally, but these details typically do not need to be visible to the library user. Like FreshML, Cαml provides forms of binding beyond binding a single variable; for example, its binding specifications can describe pattern-matching and **letrec** constructs.

6.2.5 *FreshLib.* Cheney [2005c] developed FreshLib, a library for Haskell that employs advanced generic programming techniques to provide nominal abstract syntax for Haskell programs. Moreover, FreshLib provides common operations such as capture-avoiding substitution and free-variables functions as generic operations. FreshLib also provides a richer family of binding structures, as well as a type class-based interface which permits users to define their own binding structures (such as

pattern matching binders). Since Haskell is purely functional, FreshLib code that performs fresh name generation has to be encapsulated in a monad.

7. CONCLUSIONS

Declarative programming derives much of its power from the fact that programs have a clear mathematical meaning. Name-binding and name-generation are one of many phenomena which seem to motivate abandoning declarativity in favor of expediency in practical Prolog programming. On the other hand, techniques for programming with names and binding based on higher-order abstract syntax introduce significant complexity, yet provide an interface that is too high-level for some situations. As a result both first-order and higher-order logic programs often depart from the declarative ideal when we wish to program with names and binding, since implementations often must depart significantly from a paper specification.

This paper investigates logic programming based on *nominal logic*. Nominal logic programs can be used to define a wide variety of computations involving names, binding, and name generation declaratively. It provides many of the benefits of higher-order abstract syntax, particularly built-in handling of renaming and α-equivalence, while still providing names as ordinary data. As a result, nominal logic programs are frequently direct transcriptions of what one would write "on paper".

In this paper we have presented a variety of examples of nominal logic programs, thoroughly investigated the semantics of nominal logic programming, and presented some applications of the semantics. This work provides a foundation for a number of interesting future directions, including efficient constraint solving and compilation for nominal logic programs, adding nominal abstract syntax as "just another constraint domain" to existing, mature CLP implementations, and analyzing or proving metatheoretic properties of core languages or logics defined using nominal logic programs.

REFERENCES

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. 1991. Explicit substitutions. *Journal of Functional Programming 1,* 4, 375–416.

Abel, A. 2001. A third-order representation of the $\lambda\mu$-calculus. In *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, S. Ambler, R. Crole, and A. Momigliano, Eds. Electronic Notes in Theoretical Computer Science, vol. 58(1). Elsevier.

Cervesato, I. 1998. Proof-theoretic foundation of compilation in logic programming. In *IJCSLP*. 115–129.

Cervesato, I. and Pfenning, F. 2002. A linear logical framework. *Inf. Comput. 179*, 19–75.

Cheney, J. 2004a. The complexity of equivariant unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*. LNCS, vol. 3142. Springer-Verlag, Turku, Finland, 332–344.

Cheney, J. 2005a. Equivariant unification. In *Proceedings of the 2005 Conference on Rewriting Techniques and Applications (RTA 2005)*. Number 3467 in LNCS. Springer-Verlag, Nara, Japan, 74–89.

Cheney, J. 2005b. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*. 104–119.

Cheney, J. 2005c. Scrap your nameplate (functional pearl). In *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, B. Pierce, Ed. ACM, Tallinn, Estonia, 180–191.

Cheney, J. 2005d. A simpler proof theory for nominal logic. In *Proceedings of the 2005 Conference on Foundations of Software Science and Computation Structures (FOSSACS 2005)*. Number 3441 in LNCS. Springer-Verlag, Edinburgh, UK, 379–394.

Cheney, J. 2006a. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic 71,* 1, 299–320.

Cheney, J. 2006b. The semantics of nominal logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Lecture Notes in Computer Science, vol. 4079. 361–375.

Cheney, J. and Urban, C. 2004. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*. Number 3132 in LNCS. Springer-Verlag, St. Malo, France, 269–283.

Cheney, J. R. 2004b. Nominal logic programming. Ph.D. thesis, Cornell University, Ithaca, NY.

Cheng, A. S. K., Robinson, P. J., and Staples, J. 1991. Higher level meta programming in Qu-Prolog 3.0. In *International Conference on Logic Programming*. 285–298.

Church, A. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Clark, K. L., Robinson, P. J., and Hagen, R. 2001. Multi-threading and message communication in Qu-Prolog. *Theory and Practice of Logic Programming 1,* 3, 283–301.

Clocksin, W. F. and Mellish, C. S. 2003. *Programming in Prolog*, fifth ed. Springer-Verlag.

Darlington, J. and Guo, Y. 1994. Constraint logic programming in the sequent calculus. In *Proceedings of the 1994 Conference on Logic Programming and Automated Reasoning (LPAR 1994)*. Lecture Notes in Computer Science, vol. 822. Springer-Verlag, Kiev, Ukraine, 200–214.

Davey, B. A. and Priestley, H. A. 2002. *Introduction to Lattices and Order*. Cambridge University Press.

Dowek, G., Hardin, T., Kirchner, C., and Pfenning, F. 1998. Unification via explicit substitutions: The case of higher-order patterns. Tech. Rep. Rapport de Recherche 3591, INRIA. December.

Felty, A. 1993. Implementing tactics and tacticals in a higher-order logic programming language. *J. Automated Reasoning 11,* 1, 41–81.

Fiore, M. P., Plotkin, G. D., and Turi, D. 1999. Abstract syntax and variable binding. See Longo [1999], 193–202.

Gabbay, M. J. 2003. The π-calculus in FM. In *Thirty-five years of Automath*, F. Kamareddine, Ed. Kluwer.

Gabbay, M. J. and Cheney, J. 2004. A sequent calculus for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, H. Ganzinger, Ed. IEEE, Turku, Finland, 139–148.

Gabbay, M. J. and Pitts, A. M. 2002. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing 13*, 341–363.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–101.

Hamana, M. 2001. A logic programming language based on binding algebras. In *Proc. Theoretical Aspects of Computer Science (TACS 2001)*. Number 2215 in Lecture Notes in Computer Science. Springer-Verlag, 243–262.

Hannan, J. and Miller, D. 1988. Uses of higher-order unification for implementing program transformers. In *Proc. 5th Int. Conf. and Symp. on Logic Programming, Volume 2*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, Massachusetts, 942–959.

Hanus, M. 1991. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science 89*, 63–106.

Hanus, M. 1994. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19–20*, 583–628.

Harel, D., Kozen, D., and Tiuryn, J. 2000. *Dynamic Logic*. MIT Press.

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *Journal of the ACM 40,* 1, 143–184.

Hofmann, M. 1999. Semantical analysis of higher-order abstract syntax. See Longo [1999], 204–213.

Jaffar, J., Maher, M. J., Marriott, K., and Stuckey, P. J. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Leach, J., Nieva, S., and Rodríguez-Artalejo, M. 2001. Constraint logic programming with hereditary Harrop formulas. *Theory and Practice of Logic Programming 1,* 4 (July), 409–445.

Lloyd, J. W. 1987. *Foundations of Logic Programming.* Springer-Verlag.

Longo, G., Ed. 1999. *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science.* IEEE, IEEE Press, Washington, DC.

Mason, I. A. 1987. Hoare's logic in the LF. Tech. Rep. ECS-LFCS-87-32, University of Edinburgh.

Miller, D. 1989. A logical analysis of modules in logic programming. *Journal of Logic Programming 6,* 1–2 (January), 79–108.

Miller, D. 1990. An extension to ML to handle bound variables in data structures. In *Proceedings of the First ESPRIT BRA Workshop on Logical Frameworks.* 323–335.

Miller, D. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation 1,* 4, 497–536.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51*, 125–157.

Miller, D. and Tiu, A. 2003. A proof theory for generic judgments: extended abstract. In *Proceedings of the 18th Symposium on Logic in Computer Science (LICS 2003).* IEEE Press, 118–127.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, I-II. *Information and Computation 100,* 1 (September), 1–77.

Minamide, Y., Morrisett, J. G., and Harper, R. 1996. Typed closure conversion. In *Proc. 1996 Symposium on Principles of Programming Languages.* ACM Press, 271–283.

Mitchell, J. C. 2003. *Concepts in Programming Languages.* Cambridge University Press.

Mycroft, A. and O'Keefe, R. A. 1984. A polymorphic type system for Prolog. *Artificial Intelligence 23*, 295–307.

Nadathur, G. and Miller, D. 1998. Higher-order logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Vol. 5. Oxford University Press, Chapter 8, 499–590.

Nickolas, P. and Robinson, P. J. 1996. The Qu-Prolog unification algorithm: formalisation and correctness. *Theoretical Computer Science 169*, 81–112.

O'Hearn, P. and Pym, D. J. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic 5,* 2 (June), 215–244.

Parigot, M. 1992. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the 1992 International Conference on Logic Programming and Automated Reasoning (LPAR '92)*, A. Voronkov, Ed. Number 624 in LNAI. Springer-Verlag, 190–201.

Pasalic, E., Sheard, T., and Taha, W. 2000. DALI: An untyped CBV operational semantics and equational theoryofr datatypes with binders (technical development). Tech. Rep. CSE-00-007, Oregon Graduate Institute.

Pfenning, F. 1991. Logic programming in the LF logical framework. In *Logical Frameworks*, G. Huet and G. Plotkin, Eds. Cambridge University Press, 149–181.

Pfenning, F. and Elliott, C. 1989. Higher-order abstract syntax. In *Proceedings of the 1989 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89).* ACM Press, 199–208.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf — A meta-logical framework for deductive systems. In *Proc. 16th Int. Conf. on Automated Deduction (CADE-16)*, H. Ganzinger, Ed. Springer-Verlag LNAI 1632, Trento, Italy, 202–206.

Pitts, A. M. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation 183*, 165–193.

Pitts, A. M. and Gabbay, M. J. 2000. A metalanguage for programming with bound names modulo renaming. In *Proc. 5th Int. Conf. on Mathematics of Programme Construction (MPC2000)*, R. Backhouse and J. N. Oliveira, Eds. Number 1837 in Lecture Notes in Computer Science. Springer-Verlag, Ponte de Lima, Portugal, 230–255.

Pottier, F. 2005. An overview of Cαml. In *Proceedings of the 2005 ACM SIGPLAN Workshop on ML (ML 2005)*. ACM, Tallinn, Estonia.

Reynolds, J. C. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 55–74.

Röckl, C. 2001. A first-order syntax for the pi-calculus in Isabelle/HOL using permutations. In *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, S. Ambler, R. Crole, and A. Momigliano, Eds. Electronic Notes in Theoretical Computer Science, vol. 58(1). Elsevier.

Sangiorgi, D. and Walker, D. 2001. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press.

Schürmann, C. 2001a. Recursion for higher-order encodings. In *CSL*, L. Fribourg, Ed. Lecture Notes in Computer Science, vol. 2142. Springer, 585–599.

Schürmann, C. 2001b. A type-theoretic approach to induction with higher-order encodings. In *LPAR*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 266–281.

Schürmann, C., Poswolsky, A., and Sarnat, J. 2005. The nabla-calculus. functional programming with higher-order encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calcului and Applications (TLCA 2005)*. Number 3461 in LNCS. Springer-Verlag, Nara, Japan, 339–353.

Shinwell, M. R. and Pitts, A. M. 2005. On a monadic semantics for freshness. *Theoretical Computer Science 342*, 28–55.

Shinwell, M. R., Pitts, A. M., and Gabbay, M. J. 2003. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*. ACM Press, Uppsala, Sweden, 263–274.

Staples, J., Robinson, P. J., Paterson, R. A., Hagen, R. A., Craddock, A. J., and Wallis, P. C. 1996. Qu-Prolog: An extended Prolog for meta level programming. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. MIT Press, Chapter 23, 435–452.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

Tiu, A. 2006. A logic for reasoning about generic judgments. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP'06)*. ENTCS. Elsevier. To appear.

Urban, C. and Cheney, J. 2005. Avoiding equivariant unification. In *Proceedings of the 2005 Conference on Typed Lambda Calculus and Applications (TLCA 2005)*. Number 3461 in LNCS. Springer-Verlag, Nara, Japan, 74–89.

Urban, C., Pitts, A. M., and Gabbay, M. J. 2004. Nominal unification. *Theoretical Computer Science 323,* 1–3, 473–497.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23,* 4, 293–322.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. 2003. A concurrent logical framework: The propositional fragment. In *Proceedings of TYPES 2003*. 355–377.

Received Month Year; revised Month Year; accepted Month Year

A. PROOFS FROM SECTION 4.1

LEMMA A.1 (LEMMA 4.8). *For any program Δ, T_Δ is monotone and continuous.*

PROOF OF LEMMA 4.8. We prove by induction on the structure of D that T_D has the above properties. Monotonicity is straightforward. For continuity, let $S_0, S_1, \ldots,$ be an ω-chain of subsets of $B_{\mathcal{L}}$. The cases for $\top, \wedge, \Rightarrow, \forall, \Rightarrow$, and atomic formulas follow standard arguments. For $\mathsf{И}$,

—Suppose $D = G \Rightarrow D'$. Suppose that $A \in T_{G\Rightarrow D'}(\bigcup_i S_i)$. If $\bigcup_i S_i \vDash G$ then $A \in T_{D'}(\bigcup_i S_i)$, and by induction $A \in \bigcup_i T_{D'}(S_i) = \bigcup_i T_{G\Rightarrow D'}(S_i)$. Otherwise, $A \in \bigcup_i S_i = \bigcup_i T_{G\Rightarrow D'}(S_i)$. This shows that $T_{G\Rightarrow D'}(\bigcup_i S_i) \subseteq \bigcup_i T_{G\Rightarrow D'}(S_i)$. For the reverse direction, suppose $A \in \bigcup_i T_{G\Rightarrow D'}(S_i)$. Then for some i, $A \in T_{G\Rightarrow D'}(S_i)$. There are two cases. If $S_i \vDash G$, then $A \in T_{D'}(S_i) = T_{G\Rightarrow D'}(S_i) \subseteq T_{G\Rightarrow D'}(\bigcup_i(S_i))$. Otherwise, $A \in S_i = T_{G\Rightarrow D'}(S_i) \subseteq T_{G\Rightarrow D'}(\bigcup_i(S_i))$.

—Suppose $D = \mathsf{И}\mathsf{a}.D'$. Then we have

$$\begin{array}{rcll} T_{\mathsf{И}\mathsf{a}.D'}(\bigcup_i S_i) & = & \bigcup_{\mathsf{b}:\nu\notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\bigcup_i S_i) & \text{Definition} \\ & = & \bigcup_{\mathsf{b}:\nu\notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} \bigcup_i T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(S_i) & \text{Induction hyp.} \\ & = & \bigcup_i \bigcup_{\mathsf{b}:\nu\notin \mathrm{supp}(\mathsf{И}\mathsf{a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(S_i) & \text{Unions commute} \\ & = & \bigcup_i T_{\mathsf{И}\mathsf{a}.D'}(S_i) & \text{Definition} \end{array}$$

This completes the proof. □

LEMMA A.2 (LEMMA 4.9). *For any $\mathsf{a}, \mathsf{b} \in \mathbb{A}$, $(\mathsf{a}\ \mathsf{b}) \cdot T_D(S) = T_{(\mathsf{a}\ \mathsf{b})\cdot D}((\mathsf{a}\ \mathsf{b})\cdot S)$. In particular, if Δ is a closed program with $FV(\Delta) = \mathrm{supp}(\Delta) = \varnothing$, then T_Δ is equivariant.*

PROOF OF LEMMA 4.9. The proof is by induction on the structure of D. The cases for $\top, A, \wedge$ are straightforward; for $\Rightarrow$ we need the easy observation that $S \vDash G \iff (\mathsf{a}\ \mathsf{b})\cdot S \vDash (\mathsf{a}\ \mathsf{b})\cdot G$. For $\forall X{:}\sigma.D$ formulas, observe that

$$\begin{array}{rcll} (\mathsf{a}\ \mathsf{b})\cdot T_{\forall X.D}(S) & = & (\mathsf{a}\ \mathsf{b})\cdot \bigcup_{t:\sigma} T_{D[t/X]}(S) & \text{Definition} \\ & = & \bigcup_{t:\sigma} (\mathsf{a}\ \mathsf{b}) \cdot T_{D[t/X]}(S) & \text{Swapping commutes with union} \\ & = & \bigcup_{t:\sigma} T_{((\mathsf{a}\ \mathsf{b})\cdot D)[(\mathsf{a}\ \mathsf{b})\cdot t/X]}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Induction hyp.} \\ & = & \bigcup_{u:\sigma} T_{((\mathsf{a}\ \mathsf{b})\cdot D)[u/X]}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Change of variables } (u = (\mathsf{a}\ \mathsf{b}) \cdot t) \\ & = & T_{(\mathsf{a}\ \mathsf{b})\cdot \forall X.D}((\mathsf{a}\ \mathsf{b})\cdot S) & \text{Definition.} \end{array}$$

For $\mathsf{И}$, the argument is similar. □

LEMMA A.3 (LEMMA 4.10). *If $\mathcal{M}$ is a fixed point of T_Δ, then $\mathcal{M} \vDash \Delta$.*

PROOF OF LEMMA 4.10. We first prove by induction on the structure of D that if $T_D(\mathcal{M}) = \mathcal{M}$ then $\mathcal{M} \vDash D$.

—If $D = \top$, trivially $\mathcal{M} \vDash \top$.

—If $D = A$, then clearly $\mathcal{M} \cup \{A\} = T_A(\mathcal{M}) = \mathcal{M}$ implies $A \in \mathcal{M}$ so $\mathcal{M} \vDash A$.

—If $D = D_1 \wedge D_2$, then $T_{D_1\wedge D_2}(\mathcal{M}) = T_{D_1}(\mathcal{M}) \cup T_{D_2}(\mathcal{M}) = \mathcal{M}$ implies $T_{D_1}(\mathcal{M}) = T_{D_2}(\mathcal{M}) = \mathcal{M}$ since T_{D_1}, T_{D_2} are monotone. Then using the induction hypothesis $\mathcal{M} \vDash D_1$ and $\mathcal{M} \vDash D_2$, so $\mathcal{M} \vDash D_1 \wedge D_2$.

—If $D = G \Rightarrow D'$, suppose that $\mathcal{M} \vDash G$. Then $T_{G\Rightarrow D'}(\mathcal{M}) = T_{D'}(\mathcal{M}) = \mathcal{M}$ so by induction $\mathcal{M} \vDash D'$. Hence $\mathcal{M} \vDash G \Rightarrow D'$.

—For $D = \forall X{:}\sigma.D'$, note that $\mathcal{M} = T_{\forall X.D'}(\mathcal{M}) = \bigcup_{t:\sigma} T_{D'[t/X]}(\mathcal{M})$ implies $T_{D'[t/X]}(\mathcal{M}) = \mathcal{M}$ for every $t : \sigma$. Hence by the induction hypothesis $\mathcal{M} \vDash D'[t/X]$ for every $t : \sigma$; consequently $M \vDash \forall X.D'$.

—For $D = \mathsf{И a}{:}\nu.D'$, note that $\mathcal{M} = T_{\mathsf{И a}.D'}(\mathcal{M}) = \bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M})$ implies $T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) = \mathcal{M}$ for every fresh b. Hence by the induction hypothesis $\mathcal{M} \vDash (\mathsf{a}\ \mathsf{b}) \cdot D'$ for every fresh b; consequently $M \vDash \mathsf{И a}.D'$.

Since any program $\Delta = \{D_1, \ldots, D_n\}$ is equivalent to a D-formula conjunction $D = D_1 \wedge \cdots \wedge D_n$, the desired result follows immediately. □

Lemma A.4 (Lemma 4.11). *If $\mathcal{M} \vDash \Delta$ then $\mathcal{M}$ is a fixed point of T_Δ.*

Proof of Lemma 4.11. Since T_Δ is monotone it suffices to show that $\mathcal{M}$ is a pre-fixed point. We first prove that for any D, if $\mathcal{M} \vDash D$ then $T_D(\mathcal{M}) \subseteq \mathcal{M}$, by induction on the structure of D.

—If $D = \top$, clearly $T_\top(\mathcal{M}) = \mathcal{M}$.

—If $D = A$ then since $\mathcal{M} \vDash A$, we must have $A \in \mathcal{M}$, so $T_A(\mathcal{M}) = \mathcal{M} \cup \{A\} = \mathcal{M}$.

—If $D = D_1 \wedge D_2$, then $T_{D_1 \wedge D_2}(\mathcal{M}) = T_{D_1}(\mathcal{M}) \cup T_{D_2}(\mathcal{M}) \subseteq \mathcal{M}$ since $T_{D_i}(\mathcal{M}) \subseteq \mathcal{M}$ by induction for $i = 1, 2$.

—For $D = G \Rightarrow D'$, since by assumption $\mathcal{M} \vDash G \Rightarrow D$, there are two cases. If $\mathcal{M} \vDash G$, then $\mathcal{M} \vDash D$, and by induction $T_{G \Rightarrow D}(\mathcal{M}) = T_D(\mathcal{M}) \subseteq \mathcal{M}$. On the other hand, if $\mathcal{M} \nvDash G$, then $T_{G \Rightarrow D}(\mathcal{M}) = \mathcal{M}$.

—For $D = \forall X{:}\sigma.D'$, by assumption $\mathcal{M} \vDash \forall X{:}\sigma.D'$ so we must have $\mathcal{M} \vDash D[t/X]$ for all $t{:}\sigma$. By induction $T_{D'[t/X]}(\mathcal{M}) \subseteq \mathcal{M}$ for any $t : \sigma$ so $\bigcup_{t:\sigma} T_{D'[t/X]}(\mathcal{M}) \subseteq \mathcal{M}$.

—If $D = \mathsf{И a}{:}\nu.D'$, by assumption $\mathcal{M} \vDash \mathsf{И a}{:}\nu.D'$ so $\mathcal{M} \vDash (\mathsf{a}\ \mathsf{b}) \cdot D'$ for any $\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.D')$. By induction $T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) \subseteq \mathcal{M}$ for any $\mathsf{b} \notin \mathrm{supp}(\mathsf{И a}.D')$ so $\bigcup_{\mathsf{b}:\nu \notin \mathrm{supp}(\mathsf{И a}.D')} T_{(\mathsf{a}\ \mathsf{b})\cdot D'}(\mathcal{M}) \subseteq \mathcal{M}$.

To prove the lemma, take $\Delta = \{D_1, \ldots, D_n\}$ and $D = D_1 \wedge \cdots \wedge D_n$. If $\mathcal{M} \vDash \Delta$, then $\mathcal{M} \vDash D$, so $T_D(\mathcal{M}) \subseteq \mathcal{M}$, whence $T_\Delta(\mathcal{M}) \subseteq \mathcal{M}$. □

B. PROOFS FROM SECTION 4.2

Theorem B.1 Soundness (Theorem 4.14).

(1) If $\Sigma : \Delta; \nabla \Longrightarrow G$ is derivable then $\Sigma : \Delta, \nabla \vDash G$.

(2) If $\Sigma : \Delta; \nabla \xrightarrow{D} G$ is derivable then $\Sigma : \Delta, D, \nabla \vDash G$.

Proof of Theorem 4.14. Induction on derivations. The only novel cases involve $\mathsf{И}$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow C}\ \mathit{con}$$

Then $\Sigma : \nabla \vDash C$ implies $\Sigma : \Delta, \nabla \vDash C$ as desired.

—Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R$$

By induction, $\Sigma : \Delta, \nabla \vDash G_1$ and $\Sigma : \Delta, \nabla \vDash G_2$, so clearly $\Sigma : \Delta, \nabla \vDash G_1 \wedge G_2$.

—Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2} \vee R_i$$

By induction, $\Sigma : \Delta, \nabla \vDash G_i$ so $\Sigma : \Delta, \nabla \vDash G_1 \vee G_2$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \exists X{:}\sigma.G} \exists R$$

By induction, $\Sigma, X : \Delta, \nabla, C \vDash G$. Appealing to Lemma 3.3, we have $\Sigma : \Delta, \nabla \vDash \exists X.C$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \mathit{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \mathit{И}\mathsf{a}.G}$$

By induction we have that $\Sigma\#\mathsf{a} : \Delta, \nabla, C \vDash G$. Appealing to Lemma 3.4, we conclude $\Sigma : \Delta, \nabla \vDash \mathit{И}\mathsf{a}.G$.

—Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A} \mathit{sel}$$

Then by induction hypothesis (2), we have that $\Sigma : \Delta, D, \nabla \vDash A$. Since $D \in \Delta$, clearly $\Sigma : \Delta \vDash D$ so we can deduce $\Sigma : \Delta, \nabla \vDash A$.

For the second part, proof is by induction on the derivation of $\Sigma : \Delta; \nabla \xrightarrow{D} G$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A} \mathit{hyp}$$

We need to show $\Sigma : \Delta, A', \nabla \vDash A$. To see this, suppose θ satisfies ∇ and $\mathcal{H}$ is an Herbrand model of $\Delta, \theta(A')$. Since $\Sigma : \nabla \vDash A' \sim A$, there must be a permutation π such that $\pi{\cdot}\theta(A') = \theta(A)$. Moreover, since $\mathcal{H} \vDash \theta(A')$, by the equivariance of $\mathcal{H}$ we also have $\mathcal{H} \vDash \pi{\cdot}\theta(A')$ so $\mathcal{H} \vDash \theta(A)$. Since θ and $\mathcal{H}$ were arbitrary, we conclude that $\Sigma : \Delta, A', \nabla \vDash A$.

—Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A} \wedge L_i$$

By induction, we know that $\Sigma : \Delta, D_i, \nabla \vDash A$, so can conclude $\Sigma : \Delta, D_1 \wedge D_2, \nabla \vDash A$ by Lemma 3.5.

—Suppose we have derivation

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A} \Rightarrow L$$

Then by induction, we have that $\Sigma : \Delta, D, \nabla \vDash A$ and $\Sigma : \Delta, \nabla \vDash G$. Then we can conclude $\Sigma : \Delta, G \Rightarrow D, \nabla \vDash A$ using Lemma 3.6.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A}\ \forall L$$

Then by induction, we have that $\Sigma, X : \Delta, D, \nabla, C \vDash A$. Want to conclude that $\Sigma : \Delta, \forall X.D, \nabla \vDash A$. Suppose $\Sigma : \theta \vDash \nabla$. Since $\Sigma : \nabla \vDash \exists X.C$, we have that $\Sigma : \theta \vDash \exists X.C$. Thus, there exists a t such that $\Sigma, X : \theta[X \mapsto t] \vDash C$. Therefore, $\Sigma, X : \Delta, D, \theta[X \mapsto t] \vDash A$. Since X appears only in D, by Lemma 3.7, we have that $\Sigma : \Delta, \forall X.D, \theta \vDash A$. Since θ was an arbitrary valuation satisfying ∇, it follows that $\Sigma : \Delta, \forall X.D, \nabla \vDash A$.

—Suppose we have derivation

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\text{И}\mathsf{a}.D} A}$$

By induction, we know that $\Sigma\#\mathsf{a} : \Delta, D, \nabla, C \vDash A$. Since $\Sigma : \nabla \vDash \text{И}\mathsf{a}.C$ it follows that $\Sigma\#\mathsf{a} : \Delta, D, \nabla, C \vDash A$, so by Lemma 3.2 we have $\Sigma\#\mathsf{a} : \Delta, D, \nabla \vDash A$. Moreover, by Lemma 3.8, we can conclude $\Sigma : \Delta, \text{И}\mathsf{a}.D, \nabla \vDash A$.

This completes the proof. □

Proposition B.2 (Proposition 4.15). *For any $\Sigma, \Delta, G, D, i \geq 0$:*

(1) *If $\Sigma : T^i_\Delta, \theta \vDash G$ then there exists ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma : \Delta; \nabla \Longrightarrow G$ is derivable.*

(2) *If $\Sigma : T_{\theta(D)}(T^i_\Delta), \theta \vDash A$ but $\Sigma : T^i_\Delta, \theta \nvDash A$ then there exists ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma : \Delta; \nabla \xrightarrow{D} A$.*

Proof of Proposition 4.15. For the first part, proof is by induction on i and G; most cases are straightforward.

—If $G = \top$ then trivially $\Sigma : \Delta; \cdot \Longrightarrow \top$.

—If $G = C$, a constraint, then $\Sigma : T^i_\Delta, \theta \vDash C$. By definition, this means that $\vDash \theta(C)$ holds; equivalently, $\Sigma : \theta \vDash C$. Thus, taking $\nabla = C$, we obviously have

$$\frac{}{\Sigma : \Delta; C \Longrightarrow C}\ \mathit{con}\ .$$

—If $G = A$ and $i = 0$, this case is vacuous since no atomic formulas are satisfied in the empty model T^0_Δ.

—If $G = A$ and $i > 0$, then there are two further cases. If $\Sigma : T^{i-1}_\Delta, \theta \vDash A$ then we use part (1) of the induction hypothesis with $i - 1$ to conclude $\Sigma : \Delta; \nabla \Longrightarrow A$. Otherwise $\Sigma : T^{i-1}_\Delta, \theta \nvDash A$. This implies that $\theta(A) \in T_\Delta(T^{i-1}_\Delta) = \bigcup_{D \in \Delta} T_D(T^{i-1}_\Delta)$, so we must have $\theta(A) \in T_D(T^{i-1}_\Delta)$ for some $D \in \Delta$. Since $D \in \Delta$ is closed, we have $D = \theta(D)$, so $\Sigma : T_{\theta(D)}(T^{i-1}_\Delta), \theta \vDash A$ but $\Sigma : T^{i-1}_\Delta \nvDash A$. Induction hypothesis (2) applies and we can obtain a derivation of $\Sigma : \Delta; \nabla \xrightarrow{D} A$.

The following derivation completes this case:

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel\ .$$

—If $G = G_1 \wedge G_2$, then $\Sigma : T^i_\Delta, \theta \vDash G_1 \wedge G_2$ implies $\Sigma : T^i_\Delta, \theta \vDash G_1$ and $\Sigma : T^i_\Delta, \theta \vDash G_2$, so by induction for some ∇_1, ∇_2, we have $\Sigma : \Delta; \nabla_1 \Longrightarrow G_1$, $\Sigma : \theta \vDash \nabla_1$, $\Sigma : \Delta; \nabla \Longrightarrow G_2$, and $\Sigma : \theta \vDash \nabla_2$. We can therefore conclude

$$\frac{\Sigma : \Delta; \nabla_1 \wedge \nabla_2 \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla_1 \wedge \nabla_2 \Longrightarrow G_2}{\Sigma : \Delta; \nabla_1 \wedge \nabla_2 \Longrightarrow G_1 \wedge G_2}\ .$$

since clearly $\Sigma : \theta \vDash \nabla_1 \wedge \nabla_2$.

—If $G = G_1 \vee G_2$, then $\Sigma : T^i_\Delta, \theta \vDash G_1 \vee G_2$ implies $\Sigma : T^i_\Delta, \theta \vDash G_i$ for $i \in \{1, 2\}$. In either case, by induction $\Sigma : \Delta; \nabla \Longrightarrow G_i$ and $\Sigma : \theta \vDash \nabla$ hold for some ∇, so we deduce

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2}\ .$$

—If $G = \exists X{:}\sigma.G'$, then $\Sigma : T^i_\Delta, \theta \vDash \exists X{:}\sigma.G'$ implies $\Sigma, X{:}\sigma : T^i_\Delta, \theta[X \mapsto t] \vDash G'$ for some $t : \sigma$. By induction, then, there exists ∇ such that $\Sigma, X{:}\sigma : \Delta; \nabla \Longrightarrow G'$ is derivable and $\Sigma, X : \theta[X \mapsto t] \vDash \nabla$. We can therefore derive

$$\frac{\Sigma : \exists X.\nabla \vDash \exists X.\nabla \quad \Sigma, X{:}\sigma : \Delta; \exists X.\nabla, \nabla \Longrightarrow G'}{\Sigma : \Delta; \exists X.\nabla \Longrightarrow \exists X{:}\sigma.G'}$$

using weakening to obtain the second subderivation. Clearly $\Sigma, X : \theta[X \mapsto t] \vDash \nabla$ implies $\Sigma : \theta \vDash \exists X.\nabla$.

—If $G = \mathsf{И a}{:}\nu.G'$, assume without loss of generality $\mathsf{a} \notin \Sigma$. Then $\Sigma : T^i_\Delta, \theta \vDash \mathsf{И a}.G'$ implies $\Sigma \# \mathsf{a} : T^i_\Delta, \theta \vDash G'$. By induction, there exists ∇ such that $\Sigma \# \mathsf{a} : \Delta; \nabla \Longrightarrow G'$ is derivable and $\Sigma \# \mathsf{a} : \theta \vDash \nabla$. We can therefore derive

$$\frac{\Sigma : \mathsf{И a}.\nabla \vDash \mathsf{И a}.\nabla \quad \Sigma, X{:}\sigma : \Delta; \mathsf{И a}.\nabla, \nabla \Longrightarrow G'}{\Sigma : \Delta; \mathsf{И a}.\nabla \Longrightarrow \mathsf{И a}.G'}$$

using weakening to obtain the second subderivation. Clearly, $\Sigma \# \mathsf{a} : \theta \vDash \nabla$ implies $\Sigma : \theta \vDash \mathsf{И a}.\nabla$

Similarly, the second part follows by induction on D, unwinding the definition of T_D in each case.

—If $D = \top$, then $\theta(\top) = \top$ and $T_\top(S) = S$; we cannot have both $\Sigma : T^i_\Delta, \theta \vDash A$ and $\Sigma : T^i_\Delta, \theta \nvDash A$ so this case is vacuous.

—If $D = A'$, then $T_{\theta(A')}(S) = S \cup \{\theta(A')\}$. Thus, if $\Sigma : T^i_\Delta \cup \{\theta(A')\}, \theta \vDash A$ but $\Sigma : T^i_\Delta, \theta \nvDash A$, then we must have $\theta(A) = \theta(A')$. This clearly implies $\Sigma : \theta \vDash A \approx A'$, so taking $\nabla = A \approx A'$, clearly $\Sigma : \nabla \vDash A \sim A'$ and we can derive

$$\frac{\Sigma : A \approx A' \vDash A \sim A'}{\Sigma : \Delta; A \approx A' \xrightarrow{A'} A}\ .$$

—If $D = D_1 \wedge D_2$, then $\theta(D) = \theta(D_1) \wedge \theta(D_2)$, and $T_{\theta(D_1) \wedge \theta(D_2)}(S) = T_{\theta(D_1)}(S) \cup T_{\theta(D_2)}(S)$, and $\Sigma : T_{\theta(D_1)}(T^i_\Delta) \cup T_{\theta(D_2)}(T^i_\Delta), \theta \vDash A$. Then we must have $\Sigma :$

$T_{\theta(D_j)}(T^i_\Delta), \theta \vDash A$ for $j \in \{1,2\}$. In either case, by induction there exists ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma : \Delta; \nabla \xrightarrow{D_j} A$, so we can conclude

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_j} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}$$

—If $D = G \Rightarrow D'$, then $\theta(D) = \theta(G) \Rightarrow \theta(D')$. There are two cases. If $\Sigma : T^i_\Delta, \theta \vDash G$, then $T_{\theta(G)\Rightarrow\theta(D')}(T^i_\Delta) = T_{\theta(D')}(T^i_\Delta)$ so $\Sigma : T_{\theta(D')}(T^i_\Delta), \theta \vDash A$. By induction hypothesis (1), it follows that there exists a ∇ such that $\Sigma : \Delta; \nabla \Longrightarrow G$ and $\Sigma : \theta \vDash \nabla$; by induction hypothesis (2) there also exists a ∇' such that $\Sigma : \Delta; \nabla' \xrightarrow{D'} A$ and $\Sigma : \theta \vDash \nabla'$. Using weakening and the $\Rightarrow L$ rule, we conclude

$$\frac{\Sigma : \Delta; \nabla, \nabla' \Longrightarrow G \quad \Sigma : \Delta; \nabla, \nabla' \xrightarrow{D'} A}{\Sigma : \Delta; \nabla, \nabla' \xrightarrow{G \Rightarrow D'} A}$$

which suffices since $\Sigma : \theta \vDash \nabla, \nabla'$.
Otherwise, if $\Sigma : T^i_\Delta, \theta \nvDash G$, then $T_{\theta(G)\Rightarrow\theta(D')}(T^i_\Delta) = T^i_\Delta$. Then this case is vacuous since we cannot have both $\Sigma : T^i_\Delta, \theta \vDash A$ and $\Sigma : T^i_\Delta, \theta \nvDash A$.

—If $D = \forall X{:}\sigma.D'$, assume without loss of generality that $X \notin Dom(\Sigma) \cup Dom(\theta)$. Observe that $\theta(D) = \forall X{:}\sigma.\theta(D')$. Since $T_{\forall X:\sigma.\theta(D')}(S) = \bigcup_{t:\sigma} T_{\theta(D')[t/X]}(S)$, we must have $\Sigma : \bigcup_{t:\sigma} T_{\theta(D')[t/X]}(T^i_\Delta), \theta \vDash A$. Hence, there must be a $t : \sigma$ such that $\theta(A) \in T_{\theta(D')[t/X]}(T^i_\Delta)$; choose a particular $t : \sigma$. Consequently, $\Sigma : T_{\theta(D')[t/X]}(T^i_\Delta), \theta \vDash A$. Moreover, since X is not present in Σ, A, θ, this is equivalent to $\Sigma, X : T_{\theta[X\mapsto t](D')}(T^i_\Delta), \theta[X \mapsto t] \vDash A$. By induction, there must exist a ∇ such that $\Sigma, X : \theta, [X \mapsto t] \vDash \nabla$ and $\Sigma, X : \Delta; \nabla \xrightarrow{D'} A$ holds. Hence, $\Sigma : \theta \vDash \exists X.\nabla$ so we can conclude by deriving

$$\frac{\Sigma : \exists X.\nabla \vDash \exists X.\nabla \quad \Sigma, X{:}\sigma : \Delta; \exists X.\nabla, \nabla \xrightarrow{D'} A}{\Sigma : \Delta; \exists X.\nabla \xrightarrow{\forall X:\sigma.D'} A}$$

—If $D = \mathsf{И a}{:}\nu.D'$, assume without loss of generality that $\mathsf{a} \notin \Sigma, \theta, A$. Then $\theta(D) = \mathsf{И a}.\theta(D')$ and since $T_{\mathsf{И a}.\theta(D')}(S) = \bigcup_{\mathsf{b}\notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a\ b})\cdot\theta(D')}(S)$, so we must have $\Sigma : \bigcup_{\mathsf{b}\notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a\ b})\cdot\theta(D')}(T^i_\Delta), \theta \vDash A$. By definition, this means that $\theta(A) \in \bigcup_{\mathsf{b}\notin \mathrm{supp}(\mathsf{И a}.\theta(D'))} T_{(\mathsf{a\ b})\cdot\theta(D')}(T^i_\Delta)$. Since by assumption $\mathsf{a} \notin \Sigma, \theta, A$ and $\mathsf{a} \notin \mathrm{supp}(\mathsf{И a}.D')$, we must have $\theta(A) \in T_{(\mathsf{a\ a})\cdot\theta(D')}(T^i_\Delta)$. Note that $(\mathsf{a\ a})\cdot\theta(D') = \theta(D')$, and $\theta : \Sigma\#\mathsf{a}$, hence $\Sigma\#\mathsf{a} : T_{\theta(D')}(T^i_\Delta), \theta \vDash A$. Consequently, by induction, there exists a ∇ such that $\Sigma : \theta \vDash \nabla$ and $\Sigma\#\mathsf{a} : \Delta; \nabla \xrightarrow{D'} A$. Therefore, we have

$$\frac{\Sigma : \mathsf{И a}.\nabla \vDash \mathsf{И a}.\nabla \quad \Sigma\#\mathsf{a} : \Delta; \mathsf{И a}.\nabla, \nabla \xrightarrow{D'} A}{\Sigma : \Delta; \mathsf{И a}.\nabla \xrightarrow{\mathsf{И a}.D'} A}$$

Moreover, clearly $\Sigma\#\mathsf{a} : \theta \vDash \nabla$ implies $\Sigma : \theta \vDash \mathsf{И a}.\nabla$.

This exhausts all cases and completes the proof. □

Theorem B.3 Residuated Soundness (Theorem 4.19).

(1) *If* $\Sigma : \Delta \Longrightarrow G \setminus C$ *then* $\Sigma : \Delta; C \Longrightarrow G$.

(2) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *and* $\Sigma : \Delta \xrightarrow{D} A \setminus G$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$.

Proof of Theorem 4.19. Both parts are by structural induction on derivations.

—If the derivation is of the form

$$\frac{}{\Sigma : \Delta \Longrightarrow C \setminus C}\ con$$

then deriving $\Sigma : \Delta; C \Longrightarrow C$ is immediate.

—For derivation

$$\frac{\Sigma : \Delta \Longrightarrow G_1 \setminus C_1 \quad \Sigma : \Delta \Longrightarrow G_2 \setminus C_2}{\Sigma : \Delta \Longrightarrow G_1 \wedge G_2 \setminus C_1 \wedge C_2}\ \wedge R$$

by induction we have $\Sigma : \Delta; C_1 \Longrightarrow G_1$ and $\Sigma : \Delta; C_2 \Longrightarrow G_2$. Weakening both sides, we have $\Sigma : \Delta; C_1 \wedge C_2 \Longrightarrow G_1$ and $\Sigma : \Delta; C_1 \wedge C_2 \Longrightarrow G_2$, so can derive

$$\frac{\Sigma : \Delta; C_1 \wedge C_2 \Longrightarrow G_1 \quad \Sigma : \Delta; C_1 \wedge C_2 \Longrightarrow G_2}{\Sigma : \Delta; C_1 \wedge C_2 \Longrightarrow G_1 \wedge G_2}$$

—For derivation

$$\frac{\Sigma : \Delta \Longrightarrow G_i \setminus C}{\Sigma : \Delta \Longrightarrow G_1 \vee G_2 \setminus C}\ \vee R_i$$

by induction we have $\Sigma : \Delta; C \Longrightarrow G_i$, so can derive

$$\frac{\Sigma : \Delta; C \Longrightarrow G_i}{\Sigma : \Delta; C \Longrightarrow G_1 \vee G_2}$$

—For derivation

$$\frac{\Sigma, X : \Delta \Longrightarrow G \setminus C}{\Sigma : \Delta \Longrightarrow \exists X{:}\sigma.G \setminus \exists X.C}\ \exists R$$

by induction, we have $\Sigma, X : \Delta; C \Longrightarrow G$. Weakening this derivation, we obtain

$$\frac{\Sigma : \exists X.C \vDash \exists X.C \quad \Sigma, X : \Delta; \exists X.C, C \Longrightarrow G}{\Sigma : \Delta; \exists X.C \Longrightarrow \exists X.G}$$

—For derivation

$$\frac{\Sigma \# \mathsf{a} : \Delta \Longrightarrow G \setminus C}{\Sigma : \Delta \Longrightarrow И\mathsf{a}.G \setminus И\mathsf{a}.C}\ И R$$

by induction, we have $\Sigma \# \mathsf{a} : \Delta; C \Longrightarrow G$. Weakening this derivation, we obtain

$$\frac{\Sigma : И\mathsf{a}.C \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; И\mathsf{a}.C, C \Longrightarrow G}{\Sigma : \Delta; И\mathsf{a}.C \Longrightarrow И\mathsf{a}.G}$$

—For derivation

$$\frac{\Sigma : \Delta \xrightarrow{D} A \setminus G \quad \Sigma : \Delta \Longrightarrow G \setminus C \quad (D \in \Delta)}{\Sigma : \Delta \Longrightarrow A \setminus C}\ back$$

by induction on the second derivation, we know that $\Sigma : \Delta; C \Longrightarrow G$ holds. By induction hypothesis (2) on the first subderivation, it follows that $\Sigma : \Delta; C \xrightarrow{D} A$ holds. Hence, since $D \in \Delta$, we can conclude

$$\frac{\Sigma : \Delta; C \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; C \Longrightarrow A}$$

For part (2), we reason simultaneously by induction on the structure of the two derivations.

—For derivations

$$\frac{}{\Sigma : \Delta \xrightarrow{A'} A \setminus A \sim A'}\ hyp \qquad \frac{\overset{\mathcal{E}}{\Sigma : \nabla \vDash A \sim A'}}{\Sigma : \Delta; \nabla \Longrightarrow A \sim A'}$$

it follows that

$$\frac{\Sigma : \nabla \vDash A \sim A'}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}$$

—For derivations

$$\frac{\overset{\mathcal{D}}{\Sigma : \Delta \xrightarrow{D_i} A \setminus G}}{\Sigma : \Delta \xrightarrow{D_1 \wedge D_2} A \setminus G}\ \wedge L_i \qquad \Sigma : \Delta; \overset{\mathcal{E}}{\nabla} \Longrightarrow G$$

by induction using $\mathcal{D}, \mathcal{E}$ we have $\Sigma : \Delta; \nabla \xrightarrow{D_i} A$ so we can conclude

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i$$

—For derivations

$$\frac{\overset{\mathcal{D}}{\Sigma : \Delta \xrightarrow{D} A \setminus G_2}}{\Sigma : \Delta \xrightarrow{G_1 \Rightarrow D} A \setminus G_1 \wedge G_2}\ \Rightarrow L \qquad \frac{\overset{\mathcal{E}_1}{\Sigma : \Delta; \nabla \Longrightarrow G_1} \quad \Sigma : \Delta; \overset{\mathcal{E}_2}{\nabla} \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}$$

By the induction hypothesis applied to $\mathcal{D}$ and $\mathcal{E}_2$, we have $\Sigma : \Delta; \nabla \xrightarrow{D} A$. Then we can conclude

$$\frac{\overset{\mathcal{E}_1}{\Sigma : \Delta; \nabla \Longrightarrow G_1} \quad \Sigma : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{G_1 \Rightarrow D} A}$$

—For derivations

$$\frac{\overset{\mathcal{D}}{\Sigma, X : \Delta \xrightarrow{D} A \setminus G'}}{\Sigma : \Delta \xrightarrow{\forall X : \sigma . D} A \setminus \exists X . G'}\ \forall L \qquad \frac{\Sigma : \nabla \vDash \exists X . C \quad \Sigma, X : \Delta; \overset{\mathcal{E}}{\nabla}, C \Longrightarrow G'}{\Sigma : \Delta; \nabla \Longrightarrow \exists X . G'}$$

we can apply the induction hypothesis applied to subderivations $\mathcal{D}, \mathcal{E}$ to obtain $\Sigma, X : \Delta; \nabla, C \xrightarrow{D} A$; hence, we can conclude

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X.D} A}$$

—For derivations

$$\frac{\overset{\mathcal{D}}{\Sigma \# \mathsf{a} : \Delta \xrightarrow{D} A \setminus G}}{\Sigma : \Delta \xrightarrow{И\mathsf{a}.D} A \setminus И\mathsf{a}.G}\ И L \qquad \frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \overset{\mathcal{E}}{\nabla}, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow И\mathsf{a}.G}$$

by induction on $\mathcal{D}, \mathcal{E}$ we can derive $\Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A$; hence we can conclude

$$\frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}.D} A}$$

This exhausts all possible cases, so the proof is complete. □

Theorem B.4 Residuated Completeness (Theorem 4.20).

(1) *If* $\Sigma : \Delta; \nabla \Longrightarrow G$ *then there exists a constraint* C *such that* $\Sigma : \Delta \Longrightarrow G \setminus C$ *and* $\Sigma : \nabla \vDash C$.
(2) *If* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *then there exists goal* G *and constraint* C *such that* $\Sigma : \Delta \xrightarrow{D} A \setminus G$ *and* $\Sigma : \Delta \Longrightarrow G \setminus C$ *and* $\Sigma : \nabla \vDash C$.

Proof of Theorem 4.20. Again, the proof is by structural induction on derivations. The main subtlety is the construction of C in each case.

—Case *con*

$$\frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow C}\ con$$

Then clearly, we immediately derive

$$\frac{}{\Sigma : \Delta \Longrightarrow C \setminus C}$$

since $\Sigma : \nabla \vDash C$.

—Case $\wedge R$

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R$$

By induction, we have C_1 such that $\Sigma : \Delta \Longrightarrow G_1 \setminus C_1$ and $\Sigma : \nabla \vDash C_1$; and C_2 such that $\Sigma : \Delta \Longrightarrow G_2 \setminus C_2$ and $\Sigma : \nabla \vDash C_2$. We can conclude that

$$\frac{\Sigma : \Delta \Longrightarrow G_1 \setminus C_1 \quad \Sigma : \Delta \Longrightarrow G_2 \setminus C_2}{\Sigma : \Delta \Longrightarrow G_1 \wedge G_2 \setminus C_1 \wedge C_2}$$

observing that $\Sigma : \nabla \vDash C_1 \wedge C_2$ follows from $\Sigma : \nabla \vDash C_1$ and $\Sigma : \nabla \vDash C_2$.

—Case $\vee R_i$

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2}\ \vee R_i$$

By induction, we have C such that $\Sigma : \Delta \Longrightarrow G_i \setminus C$ and $\Sigma : \nabla \vDash C$; we can conclude by deriving

$$\frac{\Sigma : \Delta \Longrightarrow G_i \setminus C}{\Sigma : \Delta \Longrightarrow G_1 \vee G_2 \setminus C}$$

—Case $\exists R$

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \exists X{:}\sigma.G}\ \exists R$$

By induction, we know that $\Sigma, X : \Delta \Longrightarrow G \setminus C'$ holds for some C' satisfying $\Sigma, X : \nabla, C \vDash C'$. We may derive

$$\frac{\Sigma, X : \Delta \Longrightarrow G \setminus C'}{\Sigma : \Delta \Longrightarrow \exists X.G \setminus \exists X.C'}$$

To complete this case, we need to show that $\Sigma : \nabla \vDash \exists X.C'$. This follows by Lemma 3.3.

—Case $\text{И}R$

$$\frac{\Sigma : \nabla \vDash \text{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \text{И}\mathsf{a}.G}\ \text{И}R$$

By induction, we have $\Sigma\#\mathsf{a} : \Delta \Longrightarrow G \setminus C'$ holds for some C' such that $\Sigma\#\mathsf{a} : \nabla, C \vDash C'$. We may derive

$$\frac{\Sigma\#\mathsf{a} : \Delta \Longrightarrow G \setminus C'}{\Sigma : \Delta \Longrightarrow \text{И}\mathsf{a}.G \setminus \text{И}\mathsf{a}.C'}$$

Finally, to show that $\Sigma : \nabla \vDash \text{И}\mathsf{a}.C'$, we appeal to Lemma 3.4.

—Case *sel*

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

By induction hypothesis (2), there exists C and G such that $\Sigma : \Delta \xrightarrow{D} A \setminus G$, $\Sigma : \Delta \Longrightarrow G \setminus C$ and $\Sigma : \nabla \vDash C$. Therefore, we can conclude by deriving

$$\frac{\Sigma : \Delta \xrightarrow{D} A \setminus G \quad \Sigma : \Delta \Longrightarrow G \setminus C \quad (D \in \Delta)}{\Sigma : \Delta \Longrightarrow A \setminus C}$$

Now we consider the cases arising from part (2).

—Case *hyp*

$$\frac{\Sigma : \nabla \vDash A \sim A'}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp$$

Then we take $G = A \sim A' = C$ and derive

$$\frac{}{\Sigma : \Delta \xrightarrow{A'} A \setminus A \sim A'} \qquad \frac{}{\Sigma : \Delta \Longrightarrow A \sim A' \setminus A \sim A'}$$

which suffices since $\Sigma : \nabla \vDash A \sim A'$.

—Case $\wedge L_i$

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A} \wedge L_i$$

Then, by induction, we have C and G such that $\Sigma : \Delta \xrightarrow{D_i} A \setminus G$, $\Sigma : \Delta \Longrightarrow G \setminus C$, and $\Sigma : \nabla \vDash C$. It suffices to replace the first derivation with

$$\frac{\Sigma : \Delta \xrightarrow{D_i} A \setminus G}{\Sigma : \Delta \xrightarrow{D_1 \wedge D_2} A \setminus G}$$

—Case $\Rightarrow L$

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A} \Rightarrow L$$

Then, by induction on the first subderivation, we have C' and G' such that $\Sigma : \Delta \xrightarrow{D} A \setminus G'$, $\Sigma : \Delta \Longrightarrow G' \setminus C'$, and $\Sigma : \nabla \vDash C'$. By induction on the second subderivation, we have $\Sigma : \Delta \Longrightarrow G \setminus C$ and $\Sigma : \nabla \vDash C$ for some C. To conclude, we derive

$$\frac{\Sigma : \Delta \xrightarrow{D} A \setminus G'}{\Sigma : \Delta \xrightarrow{G \Rightarrow D} A \setminus G \wedge G'} \qquad \frac{\Sigma : \Delta \Longrightarrow G \setminus C \quad \Sigma : \Delta \Longrightarrow G' \setminus C'}{\Sigma : \Delta \Longrightarrow G \wedge G' \setminus C \wedge C'}$$

since $\Sigma : \nabla \vDash C \wedge C'$ follows from $\Sigma : \nabla \vDash C$ and $\Sigma : \nabla \vDash C'$.

—Case $\forall L$

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A} \forall L$$

By induction hypothesis (2) applied to the second subderivation, there exist C' and G' such that $\Sigma, X : \Delta \xrightarrow{D} A \setminus G'$ and $\Sigma, X : \Delta \Longrightarrow G' \setminus C'$ and $\Sigma, X : \nabla, C \vDash C'$. We may therefore derive

$$\frac{\Sigma, X : \Delta \xrightarrow{D} A \setminus G'}{\Sigma : \Delta \xrightarrow{\forall X.D} A \setminus \exists X.G'} \qquad \frac{\Sigma, X : \Delta \Longrightarrow G' \setminus C'}{\Sigma : \Delta \Longrightarrow \exists X.G' \setminus \exists X.C'}$$

and conclude by observing that $\Sigma : \nabla \vDash \exists X.C'$ follows from existing assumptions by Lemma 3.3.

—Case $И L$

$$\frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}:\nu.D} A} И L$$

By induction, we can obtain a goal G and constraint C' such that $\Sigma\#\mathsf{a} : \Delta \xrightarrow{D} A \setminus G'$ and $\Sigma\#\mathsf{a} : \Delta \Longrightarrow G' \setminus C'$ and $\Sigma\#\mathsf{a} : \nabla, C \vDash C'$. Clearly, we may now derive

$$\frac{\Sigma\#\mathsf{a} : \Delta \xrightarrow{D} A \setminus G'}{\Sigma : \Delta \xrightarrow{И\mathsf{a}.D} A \setminus И\mathsf{a}.G'} \qquad \frac{\Sigma\#\mathsf{a} : \Delta \Longrightarrow G' \setminus C'}{\Sigma : \Delta \Longrightarrow И\mathsf{a}.G' \setminus И\mathsf{a}.C'}$$

To conclude, we need to verify that $\Sigma : \nabla \vDash И a.C'$. This follows by Lemma 3.4.

This exhausts all cases and completes the proof. □

C. PROOFS FROM SECTION 4.3

PROPOSITION C.1 TRANSITION SOUNDNESS (PROPOSITION 4.21). *If* $\Sigma\langle\Gamma \mid \nabla\rangle \longrightarrow \Sigma'\langle\Gamma' \mid \nabla'\rangle$ *and* $\Sigma' : \Delta \Longrightarrow \vec{G}' \setminus \vec{C}'$ *then there exist* $\vec{C}$ *such that*

(1) $\Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ *and*
(2) $\Sigma' : \nabla', \vec{C}' \vDash \nabla, \vec{C}$.

PROOF. Assume $\Sigma' : \Delta \Longrightarrow \vec{G}' \setminus \vec{C}'$ is derivable. Proof is by case decomposition on the possible transition steps.

—Case (B): If the backchaining rule is used,

$$\Sigma\langle A, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle G', \vec{G_0} \mid \nabla\rangle$$

where $\Sigma : \Delta \xrightarrow{D} A \setminus G'$ for some $D \in \Delta$, then we have $\Sigma' = \Sigma$; $\vec{G} = A, \vec{G_0}$; $\vec{G'} = G', \vec{G_0}$; $\vec{C'} = \vec{C} = C', \vec{C_0}$; and $\nabla' = \nabla$. We can extract a subderivation of $\Sigma : \Delta \Longrightarrow G' \setminus C'$ so for (1) we derive $\Sigma : \Delta \Longrightarrow A, \vec{G_0} \setminus C', \vec{C_0}$ using the *back* rule. Part (2) is trivial.

—Case (C): If the constraint rule is used, we have

$$\Sigma\langle C, \vec{G'} \mid \nabla\rangle \longrightarrow \Sigma\langle \vec{G'} \mid \nabla, C\rangle$$

where ∇, C is satisfiable. Then $\Sigma' = \Sigma; \nabla' = \nabla, C$; $\vec{G} = C, \vec{G'}$; and $\vec{C} = C, \vec{C'}$. For (1), we can derive using rule *con* $\Sigma : \Delta; C, \vec{G'} \Longrightarrow C, \vec{C'}$; part (2) is trivial.

—Case ($\top$): If the operational rule for $\top$ is used, we have

$$\Sigma\langle \top, \vec{G'} \mid \nabla\rangle \longrightarrow \Sigma\langle \vec{G'} \mid \nabla\rangle$$

Then $\Sigma' = \Sigma; \nabla' = \nabla; \vec{C} = \top, \vec{C'}$; for (1), $\Sigma : \Delta \Longrightarrow \top, \vec{G'} \setminus \top, \vec{C'}$ can be derived using $\top R$, while part (2) is trivial.

—Case ($\wedge$):

$$\Sigma\langle G_1 \wedge G_2, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle G_1, G_2, \vec{G_0} \mid \nabla\rangle$$

Then $\Sigma = \Sigma'$; $\nabla' = \nabla$; $\vec{G} = G_1 \wedge G_2, \vec{G_0}$; $\vec{G'} = G_1, G_2, \vec{G_0}$; and $\vec{C'} = C_1, C_2, \vec{C_0}$. Set $\vec{C} = C_1 \wedge C_2, \vec{C_0}$. For (1), $\Sigma : \Delta \Longrightarrow G_1 \wedge G_2, \vec{G_0} \setminus C_1 \wedge C_2, \vec{C_0}$ is derivable using $\wedge R$; moreover, for (2), observe that $\Sigma : \nabla, C_1, C_2 \vDash \nabla, C_1 \wedge C_2$.

—Case ($\vee_i$):

$$\Sigma\langle G_1 \vee G_2, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle G_i, \vec{G_0} \mid \nabla\rangle$$

Then $\Sigma = \Sigma'$; $\nabla' = \nabla$; $\vec{G} = G_1 \vee G_2, \vec{G_0}$; $\vec{G'} = G_i, \vec{G_0}$; and $\vec{C'} = C, \vec{C_0}$; so set $\vec{C} = \vec{C'}$. For (1), $\Sigma : \Delta \Longrightarrow G_1 \vee G_2, \vec{G_0} \setminus C, \vec{C_0}$ follows using $\vee R$, while (2) is trivial.

—Case ($\exists$):

$$\Sigma\langle \exists X{:}\sigma.G, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma, X{:}\sigma\langle G, \vec{G_0} \mid \nabla\rangle$$

Then $\Sigma' = \Sigma, X$; $\nabla' = \nabla$; $\vec{G} = \exists X.G, \vec{G_0}$; $\vec{G'} = G, \vec{G_0}$; $\vec{C'} = C, \vec{C_0}$, so set $\vec{C} = \exists X.C, \vec{C_0}$. We can therefore derive $\Sigma : \Delta \Longrightarrow \exists X.G, \vec{G_0} \setminus \exists X.C, \vec{C_0}$ for part (1). For part (2), we observe that $\Sigma, X : \nabla, C, \vec{C_0} \vDash \exists X.C, \vec{C_0}$.

—Case ($И$): Similar to the case for ($\exists$).

$$\Sigma\langle И\mathsf{a}{:}\nu.G, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\#\mathsf{a}{:}\nu\langle G, \vec{G_0} \mid \nabla\rangle$$

Then $\Sigma' = \Sigma\#\mathsf{a}$; $\nabla' = \nabla$, $\vec{G} = И\mathsf{a}.G, \vec{G_0}$; $\vec{G'} = G, \vec{G_0}$; $\vec{C'} = C, \vec{C_0}$, so set $\vec{C} = И\mathsf{a}.C, \vec{C_0}$. For part (1), derive $\Sigma : \Delta \Longrightarrow И\mathsf{a}.G, \vec{G_0} \setminus И\mathsf{a}.C, \vec{C_0}$ using $ИR$. For part (2), observe that $\Sigma\#\mathsf{a} : \nabla, C, \vec{C_0} \vDash И\mathsf{a}.C, \vec{C_0}$.

This completes the proof. □

Proposition C.2 Transition Completeness (Proposition 4.23). *For any nonempty $\vec{G}$ and satisfiable ∇, $\vec{C}$, if we have derivations $\vec{\mathcal{D}} :: \Sigma : \Delta \Longrightarrow \vec{G} \setminus \vec{C}$ then for some Σ', ∇', and $\vec{C'}$ we have*

(1) $\Sigma\langle \vec{G} \mid \nabla\rangle \longrightarrow \Sigma'\langle \vec{G'} \mid \nabla'\rangle$,

(2) $\mathcal{D}' :: \Sigma' : \Delta \Longrightarrow \vec{G'} \setminus \vec{C'}$, *where* $\vec{\mathcal{D}'} <^* \vec{\mathcal{D}}$

(3) $\exists\Sigma[\nabla] \vDash \exists\Sigma'[\nabla']$

Proof of Proposition 4.23. Let $\vec{G}, \vec{C}, \nabla$ be given as above. Since $\vec{G}$ is nonempty, we must have $\vec{G} = G, \vec{G_0}$ and $\vec{C} = C, \vec{C_0}$. Proof is by case decomposition of the derivation of $\Sigma : \Delta \Longrightarrow G \setminus C$.

—Suppose the derivation is of the form

$$\frac{}{\Sigma : \Delta \Longrightarrow C \setminus C}\ con$$

thus, $\vec{G} = C, \vec{G'}$ and $\vec{C} = C, \vec{C'}$. Then set $\Sigma' = \Sigma$; $\nabla' = \nabla, C$. We can take the step $\Sigma\langle C, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle \vec{G_0} \mid \nabla, C\rangle$. For (2), we already have smaller derivations $\Sigma : \Delta \Longrightarrow \vec{G_0} \setminus \vec{C_0}$ and for (3), observe that $\exists\Sigma[\nabla, (C, \vec{C_0})] \vDash \exists\Sigma[(\nabla, C), \vec{C_0}]$.

—Case $\top R$: (Special case of "con", since $\top$ is a constraint?) If the derivation is of the form

$$\frac{}{\Sigma : \Delta \Longrightarrow \top \setminus \top}\ \top R$$

then $\vec{G} = \top, \vec{G'}$ and $\vec{C} = \top, \vec{C'}$. Setting $\Sigma' = \Sigma, \nabla' = \nabla$, clearly $\Sigma\langle \top, \vec{G'} \mid \nabla\rangle \longrightarrow \Sigma\langle \vec{G'} \mid \nabla\rangle$. For (2), we already have smaller derivations $\Sigma : \Delta \Longrightarrow \vec{G'} \setminus \vec{C'}$ and for (3), $\exists\Sigma[\nabla, \top, \vec{C_0}] \vDash \exists\Sigma[\nabla, \vec{C_0}]$.

—Case $\wedge R$: If the derivation is of the form

$$\frac{\Sigma : \Delta \Longrightarrow G_1 \setminus C_1 \quad \Sigma : \Delta \Longrightarrow G_2 \setminus C_2}{\Sigma : \Delta \Longrightarrow G_1 \wedge G_2 \setminus C_1 \wedge C_2}\ \wedge R$$

Thus, $\vec{G} = G_1 \wedge G_2, \vec{G_0}$ and $\vec{C} = C_1 \wedge C_2, \vec{C_0}$. Setting $\sigma = \Sigma$; $\nabla' = \nabla$; $\vec{G'} = G_1, G_2, \vec{G_0}$; and $\vec{C} = C_1, C_2, \vec{C_0}$, we can take the operational step $\Sigma\langle G_1 \wedge G_2, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle G_1, G_2, \vec{G_0} \mid \nabla\rangle$. In addition, for (2) we have subderivations $\Sigma : \Delta \Longrightarrow G_1, G_2, \vec{G_0} \setminus C_1, C_2, \vec{C_0}$ and for (3), $\exists\Sigma[\nabla, C_1 \wedge C_2, \vec{C_0}] \vDash \exists\Sigma[\nabla, C_1, C_2, \vec{C_0}]$, as desired.

—Case $\vee R_i$: If the derivation is of the form

$$\frac{\Sigma : \Delta \Longrightarrow G_i \setminus C}{\Sigma : \Delta \Longrightarrow G_1 \vee G_2 \setminus C} \vee R_i$$

then $\vec{G} = G_1 \vee G_2$ and $\vec{C} = C, \vec{C_0}$. Setting $\Sigma' = \Sigma; \nabla' = \nabla; \vec{G'} = G_i, \vec{G_0}$; and $\vec{C'} = C, \vec{C_0}$; we can take the operational step $\Sigma\langle G_1 \vee G_2 \mid \nabla\rangle \longrightarrow \Sigma\langle G_i \mid \nabla\rangle$. Moreover, we have for part (2) immediate subderivations $\Sigma : \Delta \Longrightarrow G_i, \vec{G_0} \setminus C, \vec{C_0}$ and part (3) is trivial.

—Case $\exists R$: For a derivation of the form

$$\frac{\Sigma, X : \Delta \Longrightarrow G \setminus C}{\Sigma : \Delta \Longrightarrow \exists X{:}\sigma.G \setminus \exists X.C} \exists R$$

we have $\vec{G} = \exists X.G, \vec{G_0}$ and $\vec{C} = \exists X.C, \vec{C_0}$. Setting $\Sigma' = \Sigma, X; \nabla' = \nabla; \vec{G'} = G, \vec{G_0}; \vec{C'} = C, \vec{C_0}$; we can take the operational step $\Sigma\langle \exists X.G, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma, X\langle G, \vec{G_0} \mid \nabla\rangle$. Moreover, for part (2), from the given derivations we can obtain subderivations $\Sigma, X : \Delta \Longrightarrow G, \vec{G_0} \setminus C, \vec{C_0}$. For part (3), observe that $\exists\Sigma[\nabla, \exists X.C, \vec{C_0}] \vDash \exists\Sigma, X[\nabla, C, \vec{C_0}]$ since X is not free in $\nabla, \vec{C_0}$.

—Case $И R$: In this case, the derivation is of the form

$$\frac{\Sigma\#\mathsf{a} : \Delta \Longrightarrow G \setminus C}{\Sigma : \Delta \Longrightarrow И\mathsf{a}.G \setminus И\mathsf{a}.C} И R$$

$\vec{G} = И\mathsf{a}.G, \vec{G_0}$ and $\vec{C} = И\mathsf{a}.C, \vec{C_0}$. Setting $\Sigma' = \Sigma\#\mathsf{a}; \nabla' = \nabla; \vec{G'} = G, \vec{G_0}; \vec{C'} = C, \vec{C_0}$; we can take the operational step $\Sigma\langle И\mathsf{a}.G, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\#\mathsf{a}\langle G, \vec{G_0} \mid \nabla\rangle$. In addition, for (2) we can obtain smaller subderivations of $\Sigma\#\mathsf{a} : \Delta \Longrightarrow G, \vec{G_0} \setminus C, \vec{C_0}$ from the given derivations, and for (3) observe that $\exists\Sigma[\nabla, И\mathsf{a}.C, \vec{C_0}] \vDash \exists\Sigma\#\mathsf{a}[\nabla, C, \vec{C_0}]$ since a is not free in $\nabla, \vec{C_0}$.

—Case *back*: For a derivation of the form

$$\frac{\Sigma : \Delta \xrightarrow{D} A \setminus G' \quad \Sigma : \Delta \Longrightarrow G' \setminus C \quad (D \in \Delta)}{\Sigma : \Delta \Longrightarrow A \setminus C} \mathit{back}$$

we have $\vec{G} = A, \vec{G_0}$ and $\vec{C} = C, \vec{C_0}$. Set $\Sigma = \Sigma'$; $\vec{G'} = G', \vec{G_0}$; $\vec{C'} = C, \vec{C_0}$; $\nabla' = \nabla$. Using the first subderivation, we can take a backchaining step $\Sigma\langle A, \vec{G_0} \mid \nabla\rangle \longrightarrow \Sigma\langle G', \vec{G_0} \mid \nabla\rangle$. Moreover, for part (2), using the second subderivation we obtain a smaller derivation $\Sigma : \Delta \Longrightarrow G', \vec{G_0} \setminus C, \vec{C_0}$, and part (3) is trivial.

This completes the proof. □

D. PROOFS FROM SECTION 5.1

Theorem D.1 Correctness of elaboration (Theorem 5.1).

(1) *If* $\Delta \rightsquigarrow \Delta'$ *then* $\Sigma : \Delta; \nabla \Longrightarrow G$ *iff* $\Sigma : \Delta'; \nabla \Longrightarrow G$.

(2) *If* $\Delta \rightsquigarrow \Delta'$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *iff* $\Sigma : \Delta'; \nabla \xrightarrow{D} A$.

(3) *If* $D \rightsquigarrow D'$ *then* $\Sigma : \Delta; \nabla \xrightarrow{D} A$ *iff* $\Sigma : \Delta; \nabla \xrightarrow{D'} A$.

Proof of Theorem 5.1. Each part is a straightforward induction on derivations and case decomposition on the possible rewriting steps. Most cases are easy; we just show the appropriate derivation translations.

For (1), proof is by induction on the given derivation.

—Case *con*:

$$\frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow C}\ con \iff \frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta'; \nabla \Longrightarrow C}\ con$$

—Case $\wedge R$:

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R \iff \frac{\Sigma : \Delta'; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta'; \nabla \Longrightarrow G_2}{\Sigma : \Delta'; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R$$

—Case $\vee R_i$:

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2}\ \vee R_i \iff \frac{\Sigma : \Delta'; \nabla \Longrightarrow G_i}{\Sigma : \Delta'; \nabla \Longrightarrow G_1 \vee G_2}\ \vee R_i$$

—Case $\top R$:

$$\frac{}{\Sigma : \Delta; \nabla \Longrightarrow \top}\ \top R \iff \frac{}{\Sigma : \Delta'; \nabla \Longrightarrow \top}\ \top R$$

—Case $\exists R$:

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \exists X{:}\sigma.G}\ \exists R \iff \frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta'; \nabla, C \Longrightarrow G}{\Sigma : \Delta'; \nabla \Longrightarrow \exists X{:}\sigma.G}\ \exists R$$

—Case $И R$: If the derivation is of the form

$$\frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow И\mathsf{a}.G}\ И R \iff \frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta'; \nabla, C \Longrightarrow G}{\Sigma : \Delta'; \nabla \Longrightarrow И\mathsf{a}.G}\ И R$$

—Case *sel*: In this case, we need to consider the possible rewrite step taken on Δ. Writing D for the selected formula $D \in \Delta$, there are four possibilities:

—The rewrite step does not affect D. Hence, $D \in \Delta'$. Then we have

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D} A \quad (D \in \Delta')}{\Sigma : \Delta'; \nabla \Longrightarrow A}\ sel$$

—The rewrite step eliminates $D = \top$ from Δ. This case is vacuous because there can be no derivation with focused formula $\top$.

—The rewrite step splits $D = D_1 \wedge D_2 \in \Delta$; thus, $\Delta = \Delta_0, D_1 \wedge D_2$ and $\Delta' = \Delta, D_1, D_2$. Then we have

$$\frac{\dfrac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \quad (D_1 \wedge D_2 \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D_i} A \quad (D_i \in \Delta')}{\Sigma : \Delta'; \nabla \Longrightarrow A}\ sel$$

—The rewrite step rewrites $D \rightsquigarrow D'$; thus, $D' \in \Delta'$, and using IH(2) and IH(3) we can obtain

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D'} A \quad (D' \in \Delta')}{\Sigma : \Delta; \nabla \Longrightarrow A}\ sel$$

For (2), there are again several straightforward cases, which we illustrate using derivation transformations:

—Case *hyp*:

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp \iff \frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta'; \nabla \xrightarrow{A'} A}\ hyp$$

—Case $\wedge L_i$:

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta'; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i$$

—Case $\Rightarrow L$: Here we need both IH(1) and IH(2):

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A}\ \Rightarrow L \iff \frac{\Sigma : \Delta'; \nabla \xrightarrow{D} A \quad \Sigma : \Delta'; \nabla \Longrightarrow G}{\Sigma : \Delta'; \nabla \xrightarrow{G \Rightarrow D} A}\ \Rightarrow L$$

—Case $\forall L$:

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A}\ \forall L \iff \frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta'; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta'; \nabla \xrightarrow{\forall X:\sigma.D} A}\ \forall L$$

—Case $И L$:

$$\frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}.D} A}\ И L \iff \frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta'; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta'; \nabla \xrightarrow{И\mathsf{a}.D} A}\ И L$$

For (3), proof is by induction on the structure of derivations and of the possible rewriting steps.

We first show the easy cases involving "deep" rewriting steps.

—No rewrite rules apply to atomic D-formulas, so there are no deep rewrite cases involving *hyp*.

—Similarly, there are no rewrite rules for $\top$ so there are no deep rewrite cases involving $\top R$.

—If we have a deep rewrite step involving $\wedge$:

$$\frac{D_1 \rightsquigarrow D_1'}{D_1 \wedge D_2 \rightsquigarrow D_1' \wedge D_2}$$

then we can derive

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_1} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_1 \iff \frac{\Sigma : \Delta; \nabla \xrightarrow{D_1'} A}{\Sigma : \Delta'; \nabla \xrightarrow{D_1' \wedge D_2} A}\ \wedge L_1$$

The case for $\wedge L_2$ is symmetric.

—If we have a deep rewrite involving $\Rightarrow$:

$$\frac{D \rightsquigarrow D'}{G \Rightarrow D \rightsquigarrow G \Rightarrow D'}$$

then we proceed by induction (using IH(1) and IH(3)):

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D} A} \Rightarrow L \iff \frac{\Sigma : \Delta; \nabla \xrightarrow{D'} A \quad \Sigma : \Delta; \nabla \Longrightarrow G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D'} A} \Rightarrow L$$

—If we have a deep rewrite involving $\forall$:

$$\frac{D \rightsquigarrow D'}{\forall X.D \rightsquigarrow \forall X.D'}$$

then by induction we have

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A} \forall L \iff \frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D'} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D'} A} \forall L$$

—If we have a deep rewriting step involving $\mathsf{И}$:

$$\frac{D \rightsquigarrow D'}{\mathsf{И}\mathsf{a}.D \rightsquigarrow \mathsf{И}\mathsf{a}.D'}$$

then by induction we have

$$\frac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\mathsf{И}\mathsf{a}.D} A} \mathsf{И}L \iff \frac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D'} A}{\Sigma : \Delta; \nabla \xrightarrow{\mathsf{И}\mathsf{a}.D'} A} \mathsf{И}L$$

We now show the cases involving a basic rewrite rule applied at the head of the focused formula.

—Cases $G \Rightarrow \top \rightsquigarrow \top$, $\forall X.\top \rightsquigarrow \top$, $\mathsf{И}\mathsf{a}.\top \rightsquigarrow \top$: Any derivation of either the left-hand or right-hand derivations would contain a subderivation focused on $\top$, but there is no atomic rule focusing on $\top$, so these cases are vacuous.

—Case $D \wedge \top \rightsquigarrow D$ (and symmetrically $\top \wedge D \rightsquigarrow D$): Since there is no atomic rule focusing on $\top$, we must have

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{D \wedge \top} A} \wedge L_1 \iff \Sigma : \Delta; \nabla \xrightarrow{D} A$$

—Case $G \Rightarrow G' \Rightarrow D \rightsquigarrow G \wedge G' \Rightarrow D$: Then the left-hand derivation is

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G \qquad \dfrac{\Sigma : \Delta; \nabla \Longrightarrow G' \quad \Sigma : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{G' \Rightarrow D} A} \Rightarrow L}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow G' \Rightarrow D} A} \Rightarrow L$$

which is derivable if and only if we can also derive

$$\frac{\dfrac{\Sigma : \Delta; \nabla \Longrightarrow G \quad \Sigma : \Delta; \nabla \Longrightarrow G'}{\Sigma : \Delta; \nabla \Longrightarrow G \wedge G'} \wedge R \quad \Sigma : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{G \wedge G' \Rightarrow D} A} \Rightarrow L$$

—Case $G \Rightarrow D_1 \wedge D_2 \rightsquigarrow (G \Rightarrow D_1) \wedge (G \Rightarrow D_2)$: Then we have

$$\dfrac{\Sigma : \Delta; \nabla \Longrightarrow G \quad \dfrac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\wedge L_i}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D_1 \wedge D_2} A}\Rightarrow L \iff \dfrac{\dfrac{\Sigma : \Delta; \nabla \Longrightarrow G \quad \Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D_i} A}\Rightarrow L}{\Sigma : \Delta; \nabla \xrightarrow{(G \Rightarrow D_1) \wedge (G \Rightarrow D_2)} A}\wedge L_i$$

—Case $G \Rightarrow \forall X.D \rightsquigarrow \forall X.(G \Rightarrow D)$ where $X \notin FV(G, \Sigma)$: Then we have

$$\dfrac{\Sigma : \Delta; \nabla \overset{\mathcal{D}_1}{\Longrightarrow} G \qquad \dfrac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A \overset{\mathcal{D}_2}{}}{\Sigma : \Delta; \nabla \xrightarrow{\forall X.D} A}\forall L}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow \forall X.D} A}\Rightarrow L$$

which is derivable if and only if we can also derive

$$\dfrac{\Sigma : \nabla \vDash \forall X.C \qquad \dfrac{\Sigma, X : \Delta; \nabla, C \overset{\mathcal{D}'_1}{\Longrightarrow} G \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A \overset{\mathcal{D}_2}{}}{\Sigma, X : \Delta; \nabla, C \xrightarrow{G \Rightarrow D} A}\Rightarrow L}{\Sigma : \Delta; \nabla \xrightarrow{\forall X.(G \Rightarrow D)} A}\forall L$$

since X is not mentioned in G or ∇.

—If the rewriting step is $G \Rightarrow \mathsf{И}\mathsf{a}.D \rightsquigarrow \mathsf{И}\mathsf{a}.(G \Rightarrow D)$, where $\mathsf{a} \notin \mathrm{supp}(G, \Sigma)$, then we can derive

$$\dfrac{\Sigma : \Delta; \nabla \overset{\mathcal{D}_1}{\Longrightarrow} G \qquad \dfrac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A \overset{\mathcal{D}_2}{}}{\Sigma : \Delta; \nabla \xrightarrow{\mathsf{И}\mathsf{a}.D} A}\mathsf{И}L}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow \mathsf{И}\mathsf{a}.D} A}\Rightarrow L$$

if and only if we can also derive

$$\dfrac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \qquad \dfrac{\Sigma \# \mathsf{a} : \Delta; \nabla, C \overset{\mathcal{D}'_1}{\Longrightarrow} G \quad \Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{D} A \overset{\mathcal{D}_2}{}}{\Sigma \# \mathsf{a} : \Delta; \nabla, C \xrightarrow{G \Rightarrow D} A}\Rightarrow L}{\Sigma : \Delta; \nabla \xrightarrow{\mathsf{И}\mathsf{a}.(G \Rightarrow D)} A}\mathsf{И}L$$

since a is not mentioned in G or ∇.

—If the rewriting step is $\forall X.(D_1 \wedge D_2) \rightsquigarrow \forall X.D_1 \wedge \forall X.D_2$ then (for $i \in \{1, 2\}$) we can derive

$$\dfrac{\Sigma : \nabla \vDash \forall X.C \quad \dfrac{\Sigma, X : \Delta; \nabla, C \overset{\mathcal{D}}{\xrightarrow{D_i}} A}{\Sigma, X : \Delta; \nabla, C \xrightarrow{D_1 \wedge D_2} A}\wedge L_i}{\Sigma : \Delta; \nabla \xrightarrow{\forall X.(D_1 \wedge D_2)} A}\forall L$$

if and only if we can derive

$$\dfrac{\dfrac{\mathcal{D}}{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D_i} A}\ \forall L}{\dfrac{\Sigma : \Delta; \nabla, C \xrightarrow{\forall X.D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X.D_1 \wedge \forall X.D_2} A}\ \wedge L_i}$$

—If the rewriting step is $И\mathsf{a}.(D_1 \wedge D_2) \rightsquigarrow И\mathsf{a}.D_1 \wedge И\mathsf{a}.D_2$ then for $i \in \{1, 2\}$ we can derive

$$\dfrac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \dfrac{\dfrac{\mathcal{D}}{\Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D_i} A}\ \wedge L_i}{\Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D_1 \wedge D_2} A}\ И L}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}.(D_1 \wedge D_2)} A}$$

if and only if we can derive

$$\dfrac{\dfrac{\mathcal{D}}{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{D_i} A}\ И L}{\dfrac{\Sigma\#\mathsf{a} : \Delta; \nabla, C \xrightarrow{И\mathsf{a}.D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{И\mathsf{a}.D_1 \wedge И\mathsf{a}.D_2} A}\ \wedge L_i}$$

This completes the proof. □

E. PROOFS FROM SECTION 5.2

LEMMA E.1 (LEMMA 5.5). *Let Δ be a И-goal program and π be a type-preserving permutation of names in Σ.*

(1) If $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G$ then $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot G$.

(2) If $\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A$ then $\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} \pi \cdot A$.

PROOF OF LEMMA 5.5. By induction on derivations.

—For case *con*, we transform derivations as follows:

$$\dfrac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} C}\ con \longmapsto \dfrac{\Sigma : \nabla \vDash \pi \cdot C}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot C}\ con$$

since $\Sigma : \nabla \vDash C$ implies $\Sigma : \nabla \vDash \pi \cdot C$.

—For case $\top R$, we transform

$$\dfrac{}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \top}\ \top R \longmapsto \dfrac{}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot \top}\ \top R$$

since $\pi \cdot \top = \top$.

—For case $\wedge R$, note that $\pi \cdot (G_1 \wedge G_2) = \pi \cdot G_1 \wedge \pi \cdot G_2$, so we transform

$$\dfrac{\overset{\mathcal{D}_1}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1} \quad \overset{\mathcal{D}_2}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_2}}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1 \wedge G_2}\ \wedge R \longmapsto \dfrac{\overset{\mathcal{D}'_1}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot G_1} \quad \overset{\mathcal{D}'_2}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot G_2}}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot (G_1 \wedge G_2)}\ \wedge R$$

where by induction $\mathcal{D}_i :: \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_i \longmapsto \mathcal{D}'_i :: \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot G_i$ for $i \in \{1, 2\}$.

—For case $\vee R_i$ ($i \in \{1,2\}$), note that $\pi{\cdot}(G_1 \vee G_2) = \pi{\cdot}G_1 \vee \pi{\cdot}G_2$, so we have

$$\frac{\Sigma : \Delta; \nabla \overset{\mathcal{D}}{\Longrightarrow}_{\approx} G_i}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1 \vee G_2}\ \vee R_i \longmapsto \frac{\Sigma : \Delta; \nabla \overset{\mathcal{D}'}{\Longrightarrow}_{\approx} \pi{\cdot}G_i}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}(G_1 \vee G_2)}\ \vee R_i$$

where by induction $\mathcal{D} :: \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_i \longmapsto \mathcal{D}' :: \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}G_i$

—For case $\exists R$, we have

$$\frac{\Sigma : \nabla \vDash \exists X.C[X] \quad \Sigma, X : \Delta; \nabla, \overset{\mathcal{D}}{C[X]} \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \exists X.G}$$

Note that $\pi{\cdot}\exists X.G[X] = \exists X.\pi{\cdot}G[\pi^{-1}{\cdot}X]$. By induction,

$$\Sigma, X : \Delta; \nabla, \overset{\mathcal{D}}{C[X]} \Longrightarrow_{\approx} G \longmapsto \Sigma, X : \Delta; \nabla, \overset{\mathcal{D}'}{C[X]} \Longrightarrow_{\approx} \pi{\cdot}G[X]\ .$$

Since π is invertible, we can substitute $Y = \pi{\cdot}X$ to obtain $\mathcal{D}'' :: \Sigma, Y : \Delta; \nabla, C[\pi^{-1}{\cdot}Y] \Longrightarrow_{\approx} \pi{\cdot}G[\pi^{-1}{\cdot}Y]$; moreover, clearly, $\Sigma : \nabla \vDash \exists Y.C[\pi^{-1}{\cdot}Y]$, so we can conclude

$$\frac{\Sigma : \nabla \vDash \exists Y.C[\pi^{-1}{\cdot}Y] \quad \Sigma, Y : \Delta; \nabla, C[\pi^{-1}{\cdot}Y] \overset{\mathcal{D}''}{\Longrightarrow}_{\approx} \pi{\cdot}G[\pi^{-1}{\cdot}Y]}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}\exists X.G}\ .$$

—For case $И R$, we have derivation

$$\frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, \overset{\mathcal{D}}{C} \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} И\mathsf{a}{:}\nu.G}\ И R \longmapsto \frac{\Sigma : \nabla \vDash И\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, \overset{\mathcal{D}'}{C} \Longrightarrow_{\approx} \pi{\cdot}G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}(И\mathsf{a}{:}\nu.G)}\ И R$$

since $\pi{\cdot}И\mathsf{a}{:}\nu.G = И\mathsf{a}{:}\nu.\pi{\cdot}G$, (since, without loss, $\mathsf{a} \notin FN(\Sigma) \cup \mathrm{supp}(\pi)$). The derivation $\mathcal{D}' :: \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow_{\approx} \pi{\cdot}G$ is obtained by induction.

—For case *sel*,

$$\frac{\overset{\mathcal{D}}{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A} \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} A}\ sel \longmapsto \frac{\overset{\mathcal{D}'}{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} \pi{\cdot}A} \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}A}\ sel$$

using induction hypothesis (2) to derive $\mathcal{D}'$ from $\mathcal{D}$, and the fact that $\pi{\cdot}D = D$ (because $D \in \Delta$ is closed).

For part (2), all cases are straightforward; cases *hyp* and $И L$ are of interest.

—Case *hyp*

$$\frac{\Sigma : \nabla \vDash A' \approx A}{\Sigma : \Delta; \nabla \xrightarrow{A'}_{\approx} A}\ hyp \longmapsto \frac{\Sigma : \nabla \vDash \pi{\cdot}A' \approx \pi{\cdot}A}{\Sigma : \Delta; \nabla \xrightarrow{\pi{\cdot}A'}_{\approx} \pi{\cdot}A}\ hyp$$

since $\Sigma : A' \approx A \vDash \pi{\cdot}A' \approx \pi{\cdot}A$.

—Case $\wedge L_i$

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i}_{\approx} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2}_{\approx} A}\ \wedge L_i \longmapsto \frac{\Sigma : \Delta; \nabla \xrightarrow{\pi{\cdot}D_i}_{\approx} \pi{\cdot}A}{\Sigma : \Delta; \nabla \xrightarrow{\pi{\cdot}(D_1 \wedge D_2)}_{\approx} \pi{\cdot}A}\ \wedge L_i$$

since $\pi{\cdot}(D_1 \wedge D_2) = \pi{\cdot}D_1 \wedge \pi{\cdot}D_2$. The subderivations are constructed by induction.

—Case $\Rightarrow L$

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \xrightarrow{G \Rightarrow D}_{\approx} A} \Rightarrow L \longmapsto \frac{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} \pi \cdot A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi \cdot G}{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot (G \Rightarrow D)}_{\approx} \pi \cdot A} \Rightarrow L$$

where the subderivations are obtained by induction; this suffices because $\pi \cdot (G \Rightarrow D) = \pi \cdot G \Rightarrow \pi \cdot D$.

—Case $\forall L$: We have

$$\frac{\Sigma : \nabla \vDash \exists X.C[X] \quad \Sigma, X : \Delta; \nabla, C[X] \xrightarrow{D[X]}_{\approx} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D}_{\approx} A} \forall L$$

The argument is similar to that for $\exists R$ for part (1). By induction we have $\Sigma, X : \Delta; \nabla, C[X] \xrightarrow{\pi \cdot D[X]}_{\approx} \pi \cdot A$. Substituting $Y = \pi \cdot X$, we have $\Sigma, Y : \Delta; \nabla, C[\pi^{-1} \cdot Y] \xrightarrow{\pi \cdot D[\pi^{-1} \cdot Y]}_{\approx} \pi \cdot A$. and $\Sigma : \nabla \vDash \exists Y.C[\pi^{-1} \cdot Y]$. It follows that

$$\frac{\Sigma : \nabla \vDash \exists Y.C[\pi^{-1} \cdot Y] \quad \Sigma, Y : \Delta; \nabla, C[\pi^{-1} \cdot Y] \xrightarrow{\pi \cdot D[\pi^{-1} \cdot Y]}_{\approx} \pi \cdot A}{\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot \forall Y:\sigma.D}_{\approx} \pi \cdot A} \forall L$$

since $\pi \cdot \forall Y{:}\sigma.D = \forall Y.\pi \cdot D[\pi^{-1} \cdot Y]$.

—The case for $И L$ is vacuous because no formulas $И a.D$ can appear in a $И$-goal program.

This completes the proof. □

Theorem E.2 (Theorem 5.6). *If Δ is $И$-goal then*

(1) *If $\Sigma : \Delta; \nabla \Longrightarrow G$ is derivable, then $\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G$ is derivable.*

(2) *If $\Sigma : \Delta; \nabla \xrightarrow{D} A$ is derivable, there exists a π such that $\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} A$ is derivable.*

Proof of Theorem 5.6. The proof is by induction on derivations. For part (1), the most interesting cases is *sel*; the rest are straightforward.

—For *sel*, we have

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D} A}{\Sigma : \Delta; \nabla \Longrightarrow A}$$

for some closed $D \in \Delta$. By induction hypothesis (2), for some π, $\Sigma : \Delta; \nabla \xrightarrow{\pi \cdot D}_{\approx} A$ holds. However, since D is closed, $\pi \cdot D = D \in \Delta$ so we may conclude

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D}_{\approx} A \quad (D \in \Delta)}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} A} \; sel$$

—Case *con*:

$$\frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow C} \; con \longmapsto \frac{\Sigma : \nabla \vDash C}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} C} \; con$$

—Case $\top R$:

$$\frac{}{\Sigma : \Delta; \nabla \Longrightarrow \top}\ \top R \longmapsto \frac{}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \top}\ \top R$$

—Case $\wedge R$:

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow G_2}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \wedge G_2}\ \wedge R \longmapsto \frac{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1 \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_2}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1 \wedge G_2}\ \wedge R$$

—Case $\vee R_i$ $(i \in \{1, 2\})$:

$$\frac{\Sigma : \Delta; \nabla \Longrightarrow G_i}{\Sigma : \Delta; \nabla \Longrightarrow G_1 \vee G_2}\ \vee R_i \longmapsto \frac{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_i}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} G_1 \vee G_2}\ \vee R_i$$

—Case $\exists R$:

$$\frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \exists X{:}\sigma.G}\ \exists R \longmapsto \frac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \exists X{:}\sigma.G}\ \exists R$$

—Case $\mathsf{И}R$:

$$\frac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow G}{\Sigma : \Delta; \nabla \Longrightarrow \mathsf{И}\mathsf{a}{:}\nu.G}\ \mathsf{И}R \longmapsto \frac{\Sigma : \nabla \vDash \mathsf{И}\mathsf{a}.C \quad \Sigma\#\mathsf{a} : \Delta; \nabla, C \Longrightarrow_{\approx} G}{\Sigma : \Delta; \nabla \Longrightarrow_{\approx} \mathsf{И}\mathsf{a}{:}\nu.G}\ \mathsf{И}R$$

For part (2), the interesting cases are *hyp* and $\mathsf{И}L$.

—For *hyp*, we have

$$\frac{\Sigma : \nabla \vDash A' \sim A}{\Sigma : \Delta; \nabla \xrightarrow{A'} A}\ hyp$$

By definition $\Sigma : \nabla \vDash A' \sim A$ means there exists a π such that $\Sigma : \nabla \vDash \pi{\cdot}A' \approx A$, so

$$\frac{\Sigma : \nabla \vDash \pi{\cdot}A' \approx A}{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot A'}_{\approx} A}\ hyp$$

—Case $\wedge L_i$:

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{D_i} A}{\Sigma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \longmapsto \frac{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot D_i}_{\approx} A}{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot D_1 \wedge \pi\cdot D_2}_{\approx} A}\ \wedge L_i$$

—Case $\Rightarrow L$: Suppose we have Using both induction hypotheses, and then Lemma 5.5, we can obtain derivations of the following judgments:

$$\begin{array}{ll} \Sigma : \Delta; \nabla \xrightarrow{\pi\cdot G \Rightarrow \pi\cdot D}_{\approx} A & \text{for some } \pi\text{, by part (1)} \\ \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G & \text{by part (2)} \\ \Sigma : \Delta; \nabla \Longrightarrow_{\approx} G & \text{by Lemma 5.5} \end{array}$$

so we can conclude

$$\frac{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot D}_{\approx} A \quad \Sigma : \Delta; \nabla \Longrightarrow_{\approx} \pi{\cdot}G}{\Sigma : \Delta; \nabla \xrightarrow{\pi\cdot G \Rightarrow \pi\cdot D}_{\approx} A}\ \Rightarrow L$$

—Case $\forall L$: Using the induction hypothesis, and changing variables ($Y = \pi{\cdot}X$), we have

$$\frac{\dfrac{\Sigma : \nabla \vDash \exists X.C \quad \Sigma, X : \Delta; \nabla, C \xrightarrow{D} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall X:\sigma.D} A}\ \forall L}{\longmapsto \quad \dfrac{\Sigma : \nabla \vDash \exists Y.C[\pi^{-1}{\cdot}Y] \quad \Sigma, Y : \Delta; \nabla, C[\pi^{-1}{\cdot}Y] \xrightarrow{\pi\cdot D[\pi^{-1}\cdot Y]}_{\approx} A}{\Sigma : \Delta; \nabla \xrightarrow{\forall Y:\sigma.\pi\cdot D[\pi^{-1}\cdot Y]}_{\approx} A}\ \forall L}$$

—Case $\mathsf{И}L$ is vacuous, since no instance of $\mathsf{И}L$ can occur in a derivation involving a $\mathsf{И}$-goal program.

This completes the proof. □