NOMINAL LOGIC PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

James Robert Cheney

August 2004

NOMINAL LOGIC PROGRAMMING

James Robert Cheney, Ph.D.
Cornell University 2004

Names, name-binding, and equivalence of expressions up to "safe" renaming of bound names are often encountered in programming tasks, especially in writing programs that manipulate symbolic data such as mathematical expressions or other programs. Most programming languages provide no support for programming with names and binding, so operations like renaming and substitution must be written by hand, on an application-by-application basis.

Recently, a new theory of abstract syntax with names and binding has been developed by Gabbay and Pitts, based on defining concepts like name-binding and substitution in terms of more primitive concepts including *name-swapping* and *freshness*. We call this approach *nominal abstract syntax*. Pitts has developed a variant of first-order logic, called *nominal logic*, which formalizes nominal abstract syntax. This dissertation investigates *nominal logic programming*, or logic programming using nominal logic.

The contributions of this dissertation are as follows. A nominal logic programming language called αProlog is presented, along with examples of programs that are particularly convenient to write in αProlog. A revised form of nominal logic that provides a suitable foundation for nominal logic programming is developed. Both operational and denotational semantics for nominal logic programming are given, and soundness and completeness properties are proved. The unification and other nominal constraint problems that must be solved during execution of nominal logic programs are identified, and the complexity of and algorithms for solving these constraints are investigated.

In addition, nominal abstract syntax is compared with other advanced techniques for programming with names and binding, notably *higher-order abstract syntax* (HOAS). It is argued that nominal abstract syntax is both simpler and more expressive than HOAS: in particular, HOAS cannot easily handle *open terms*, or terms with an unknown number of free names. In nominal abstract syntax, names are explicit data, and there is no problem with open terms.

These contributions support the contention that nominal logic programming is a powerful technique for programming with names and binding.

Biographical Sketch

James Cheney was born in Brookfield, Wisconsin, and lived in southeastern Wisconsin until 1994. In that year, James was awarded the rank of Eagle Scout.

In 1994, James matriculated at Carnegie Mellon University, Pittsburgh, Pennsylvania. He received a B. S. in Computer Science (May 1998) and M.S. in Mathematics (August 1998) from CMU, graduating with a grade point average of 4.0 out of 4.0, and was inducted into the Pi Mu Epsilon, Phi Beta Kappa, and Phi Kappa Phi honor societies.

In 1998, James began graduate study at Cornell University, Ithaca, New York, pursuing research in diverse areas including structured and semi-structured data compression, digital preservation, generic programming, and type-safe imperative languages. During June and July 2002, James attended the first Proofs as Programs Summer School, held at the University of Oregon. During the spring of 2003, he visited the University of Cambridge, United Kingdom. There James became interested in the subject of this dissertation.

Awake! for Morning in the Bowl of Night
Has cast the Stone which put the Stars to Flight,
  And lo! the Hunter of the East has Caught
The Sultan's Turret in a Noose of Light.

—*Rubáiyát of Omar Khayyám*, trans. E. FitzGerald

Acknowledgements

I first wish to acknowledge my advisor, Greg Morrisett. His support for me and this work has been unyielding. My other committee members, Robert Constable and Geri Gay, have also been very supportive.

I wish to thank Steve Chong, Amy Gale, Dan Grossman, Kevin Hamlen, Evera Lovelace, Stephanie Weirich, and Steve Zdancewic for their friendship during my time at Cornell. Too many others to name also contributed to the preservation of my sanity during the often long and humid Ithaca summers. (As a native of Wisconsin, the winters seldom impressed me.)

None of this work would have been possible without the prior efforts of M. J. (Jamie) Gabbay, Andrew Pitts, and Christian Urban. All three have been very enthusiastic in their support. Andrew and Christian provided a great deal of useful feedback on drafts of this dissertation. I am particularly indebted to Christian for permission to use the name "αProlog."

Finally, I wish to thank Dexter Kozen, with whom I discussed most of this work on a weekly basis over the last few months and whose feedback has been invaluable, to the extent that I feel guilty not having added him to my committee.

This work was supported by the following grants: AFOSR F49620-01-1-0298, ONR N00014-01-1-0968, NSF 9875536, and AFOSR F49620-03-1-0156,

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

If the only tool you have is a hammer, you tend to see every problem as a nail.

—*Abraham Maslow*

1.1 The Problem of Variable Binding

In mathematical expressions such as $x + 1$, the identifier x denotes an unknown quantity. Many mathematical problems are formulated as searching for values for the identifiers in an equation that make the equation true. For example, the equation $x + 1 = 5$ has solution $x = 4$ because $4 + 1 = 5$. However, sometimes identifiers are not used to denote a specific unknown quantity, but in a more schematic way as a placeholder within an expression. Here are two examples:

$$f(x) = x + 1 \tag{1.1}$$

$$\sum_{y=1}^{10} y \tag{1.2}$$

Both these notations support a *substitution reading* of the identifiers, since we interpret $f(x) = x + 1$ as shorthand for infinitely many assertions of the form $f(1) = 2$, $f(2) = 3$, etc., and $\sum_{y=0}^{10} y = 1 + 2 + \cdots + 10$, the result of substituting 1 through 10 for y and summing the resulting sequence. We characterize the substitution reading of identifiers as follows:

> Substitution Reading: Identifiers are *variables* which stand for one or more unknown values that will later be substituted for the identifier.

In addition, both notations also support a *binding reading* of the identifiers x and y: the names x and y used are incidental to the meaning of the notations, and other identifiers would do as well. Identifiers used this way are often called *dummy variables.* Moreover, we think of the x and y in $f(x) = [\cdots]$ and $\sum_{y=0}^{10}[\cdots]$ as *bound* within the respective bracketed parts of the expression. If the larger context of the expression were

$$\sum_{y=1}^{10} y = x + y$$

we ought not to think of the occurrences of y as referring to the same thing. Instead, the occurrences on the left side are bound names whereas the occurrence on the right is unbound. We characterize the binding reading of identifiers as follows:

> Binding Reading: Identifiers are *scoped names* which may be subject to binding and "safe" renaming within their scopes.

Both the substitution and binding readings of identifiers play a particularly important role in the λ-calculus, a formal notation for writing "anonymous" functions invented by Church [20, 10]. In the term $\lambda x.t$, the variable x is considered bound in t, and x can *only* be renamed consistently to another bound variable name or substituted with another term. For example, the λ-calculus expression $\lambda x.x + 1$ describes a function which adds one to its argument, without giving the function a specific name such as f (as is done in (1.1)).

Although the substitution and binding readings of variables often appear together, some notations involve binding but not substitution. In a programming language, a local variable or function parameter x in an assignment statement such as $x := x + 1$ has a different meaning: it identifies an unspecified value that might change over time. For example, if $x = 4$, then after executing $x := x + 1$, x should be 5. Program variables also can be bound and used in a schematic way: for example, in a program logic such as *Hoare logic* or *dynamic logic* [51], we could describe the behavior of the assignment operation as follows:

$$\forall i.\forall x.\{x = i\}x := x + 1\{x = i + 1\}$$

where the formula $\{P\}S\{Q\}$ indicates that if formula P holds before statement S is executed and terminates, then Q holds afterward. While the $\forall x$ can be read as binding a name x (some other unused name y could be used instead), it cannot be read as abbreviating substitutions for values for x:

$$\{0 = 0\}0 := 0 + 1\{0 = 0 + 1\} \qquad \{1 = 0\}1 := 1 + 1\{1 = 0 + 1\} \cdots$$

since only (dynamic) program variables, not specific integers, may appear on the left of an assignment, that is, $0 := 0 + 1$ is not even a well-formed program statement.

Another example of a language in which identifiers may be bound but not substituted with other terms is the *π-calculus*, a language for describing concurrent, communicating processes, due to Milner [84]. A distinctive feature of the π-calculus is its *name-restriction* operator $\nu x.p$, which binds an identifier x within process p, and intuitively can be read as "a process with a private, local name". In contrast to the λ-calculus, a ν-bound name can only be renamed, never substituted away.

This discussion demonstrates that the substitution reading of identifiers does not apply to typical programming uses of variables or to restricted names in the π-calculus. However, the binding reading is common to all the examples. Linguistically speaking, we may be on shaky ground using the term *variable* to denote things to which the substitution reading does not apply; hence in this dissertation we shall instead use the word *name* rather than variable for identifiers to which the binding reading applies but the substitution reading may not.

While the above discussion suffices for explaining changes of bound names in summations or the behavior of local variables in programs informally in introductory courses on these subjects, it is inadequate for reasoning about languages or logics or writing programs that deal with names and binding. In order to avoid mistakes, it is necessary to define precisely when it is safe to rename a bound name. For example, in an equation such as

$$f(n) = \sum_{i=0}^{n} n \cdot i$$

it would be acceptable to rename n to m but not to i: the latter would cause confusion between "the dummy variable i in the sum" and "the value i provided as the argument to f". However, renaming n to j would be fine. The intuition is that a name can be renamed only to another name that is not already in use elsewhere.

Beginning with Frege's *Begriffsschrift* [38], the first comprehensive treatment of symbolic predicate logic, logicians have been conscious of the need for caution when dealing with name binding and substitution. In Frege's work, bound names were syntactically distinguished from unbound names: the former were written using German letters $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$, and the latter using Roman (italic) letters x, y, z. For Frege, bound names and were subject to renaming using the following principle:

> . . . Replacing a German letter everywhere in its scope by some other one is, of course, permitted, so long as in places where different letters initially stood different ones also stand afterward. This has no effect on the content. [38]

Frege did not give a completely explicit formal treatment of substitution, and as a result much of the complexity resulting from the interaction of substitution and name-binding was hidden.

These problems are put into particularly sharp focus in the λ-calculus. In a λ-expression with multiple arguments, such as $\lambda x.\lambda y.x + y$, the function that adds its first and second arguments, it seems intuitively clear that x can be renamed to z but not y, because otherwise we would have $\lambda y.\lambda y.y + y$, the function which ignores its first argument and doubles its second. This phenomenon is called variable capture, and is also familiar in first-order logic.

To prevent this kind of error, Church [20] defined the concepts of *capture-avoiding substitution*, in which bound names are renamed to avoid capture, and

α-equivalence, which permits renaming bound variables provided no capture can occur. Church was not the first to define these concepts; they had been studied by many logicians previously, with varying degrees of success. However, Church's treatment of α-equivalence and the use of capture-avoiding substitution in defining equality for λ-terms has become standard.

In informal mathematical arguments, renaming issues are often either ignored or minimized once the formal definitions have been presented. For example, the *Barendregt Variable Convention* is often taken for granted in mathematical exposition. After defining capture-avoiding substitution, renaming, and α-equivalence and proving their properties in detail, Barendregt states:

> 2.1.13. VARIABLE CONVENTION. If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g., definition, proof), then in these terms all bound variables are chosen to be different from free variables. [10]

While clear enough for human readers, however, such conventions still leave a considerable gap between mathematical exposition and correct formalizations or computer implementations of programming languages and logics involving names and binding.

1.2 Existing Approaches

There are two well-established techniques for bridging this gap: first-order and higher-order abstract syntax. In this section we discuss them in detail and sketch the state of the art of programming using these approaches. Additional techniques are discussed in Chapter 8.

As a running example, we consider the problem of implementing the λ-calculus in the various approaches. Table 1.1 shows typical *informal* definitions of $\alpha\beta\eta$-equivalence, capture-avoiding substitution, and well-formedness for simply-typed λ-terms. We assume familiarity with these concepts. The notation $FV(t)$ stands for the set of free variables of a term t.

1.2.1 First-Order Abstract Syntax

The classical approach to encoding languages involving names and binding is to model language expressions as algebraic terms, represent names using some infinite datatype such as *string*, and represent both bindings and references as concrete strings. Algebraic datatypes have a very clear and intuitive semantics based on many-sorted logic and algebraic specification [46] which supports reasoning by induction on the structure of terms.

This approach, which we term *first-order abstract syntax (FOAS)*, is facilitated by functional programming languages such as Standard ML (SML) [86] and logic programming languages such as Prolog [21] that support sophisticated programming with algebraic datatypes. For example, the λ-calculus can be encoded in SML using the following SML declaration:

Table 1.1: The simply-typed λ-calculus

Syntax:

$$\begin{array}{rcl} x & \in & Var = \{x_1, x_2, \ldots\} \\ \alpha & \in & TyVar = \{\alpha_1, \alpha_2, \ldots\} \\ t & ::= & x \mid (t\ t') \mid \lambda x.t \\ \tau & ::= & \alpha \mid \tau \to \tau' \\ \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

Capture-avoiding substitution:

$$x[t/x] = t \quad (1.3)$$

$$y[t/x] = y \qquad (y \neq x) \quad (1.4)$$

$$(t_1\ t_2)[t/x] = (t_1[t/x]\ t_2[t/x]) \quad (1.5)$$

$$(\lambda y.t')[t/x] = \lambda y.(t'[t/x]) \qquad (y \notin FV(t) \cup \{x\}) \quad (1.6)$$

Equational laws:

$$\lambda x.e \equiv_\alpha \lambda y.e[y/x] \qquad (y \notin FV(e)) \quad (1.7)$$

$$(\lambda x.e)e' \equiv_\beta e[e'/x] \quad (1.8)$$

$$e \equiv_\eta \lambda x.(e\ x) \qquad (x \notin FV(e)) \quad (1.9)$$

Well-formedness:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau \to \tau' \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (t_1\ t_2) : \tau'} \qquad \frac{\Gamma, x : \tau \vdash t : \tau' \quad (x \notin FV(\Gamma))}{\Gamma \vdash \lambda x.t : \tau \to \tau'}$$

```
datatype exp = Var of string
             | App of exp * exp
             | Lam of string * exp
```

The encoding of λ-expressions as terms is as follows:

$$\begin{aligned}\ulcorner x \urcorner &= \texttt{Var "x"} \\ \ulcorner (e_1\ e_2) \urcorner &= \texttt{App}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \\ \ulcorner \lambda x.e \urcorner &= \texttt{Lam}(\texttt{"x"}, \ulcorner e \urcorner)\end{aligned}$$

It is necessary to consider λ-terms equal "up to consistent renaming", or α-*equivalence*, as defined by the equivalence relation generated by $\equiv_\alpha$. For example, $\lambda a.a$ and $\lambda b.b$ are α-equivalent terms, but they are not represented by equal data structures: `Lam("a",Var("a"))` and `Lam("b",Var("b"))` are different terms.

Human readers can be trusted to fill in the missing α-renaming steps needed to enforce the side condition $y \notin FV(e) \cup \{x\}$ in (1.6), but computer programs cannot do so without prompting, and so to correctly implement operations like capture-avoiding substitution in a first-order encoding it is necessary to make the implicit renamings in the informal definition explicit:

```
fun subst (Var y) e x       = if x = y then e else (Var y)
  | subst (App (e1,e2)) e x = App(subst e1 e x, subst e2 e x)
  | subst (Lam (y,e')) e x  = let val z = gensym()
                                  val e'' = subst e' (Var z) y
                              in Lam(z,subst e'' e x)
                              end
```

Here, `gensym` is a procedure that generates a "fresh" string (one guaranteed not to have been used elsewhere in the program). In SML, `gensym` is implemented using references and side-effects.

Now α-equivalence can be programmed as follows:

```
fun aeq (Var x) (Var y)             = x = y
  | aeq (App(e1,e2)) (App(e1',e2')) = aeq e1 e1' andalso aeq e2 e2'
  | aeq (Lam(x, e)) (Lam(y,e'))     = let z = gensym()
                                        in aeq (subst e (Var z) x)
                                               (subst e' (Var z) y)
                                        end
```

Since binders are explicitly and aggressively renamed, even simple properties of substitution such as $subst\ e\ (Var\ x)\ x = e$ are not literally true, but only true up to α-equivalence. For example, $subst\ (Lam(\texttt{"x"}, Var\ \texttt{"x"}))\ (Var\ \texttt{"y"})\ \texttt{"y"}$ will result in $Lam(z, Var\ z)$ for some fresh string z, so is guaranteed not to be literally equal to $Lam(\texttt{"x"}, Var\ \texttt{"x"})$. Instead, we only have the weaker property $aeq(subst\ e\ (Var\ \texttt{"x"})\ \texttt{"x"})\ e$. Also, because of the use of side-effects, *subst* is not even, technically, a function, since it can return different results when called with the same arguments. For example, $subst\ (Lam(\texttt{"x"}, Var(\texttt{"x"})\ y\ z$ can produce two different values $Lam(\texttt{"z1"}, Var(\texttt{"z1"})$ and $Lam(\texttt{"z2"}, Var(\texttt{"z2"})$. This definition of substitution is only a function modulo α-equivalence.

Substitution and α-equivalence are basic operations in any program dealing with the λ-calculus, so any attempt to verify the overall correctness of such a program would be seriously handicapped by the difficulty of proving even simple facts about them [53, 129]. First-order abstract syntax is the most popular implementation technique, but it difficult to reason about languages defined using first-order syntax because of the use of side-effects[1] to generate names and because reasoning must frequently be performed "modulo α-equivalence".

Combinatory logic and de Bruijn indices are two alternative approaches to dealing with names and binding that may be called "first-order". We defer a comprehensive description of these approaches and comparison with our proposed approach to Section 8.1.1.

1.2.2 Higher-Order Abstract Syntax

An alternative, very elegant approach to specifying languages with binding structure is to encode languages within a higher-order meta-language such as the λ-calculus, model object-language variables as meta-language variables, and model object-language binding structure with meta-language λ-abstraction. Usually the meta-language is taken to be a variant of the simply-typed λ-calculus. This approach, which was first employed by Church [20] in the development of higher-order logic, is called *higher-order abstract syntax (HOAS)* [104], and is supported by logical frameworks such as Twelf [105] and Coq [32], theorem provers such as Nuprl [23] and Isabelle/HOL [96] and higher-order logic programming languages such as λProlog [90].

As an example of the use of higher-order abstract syntax, we return to the λ-calculus example discussed in the previous section. For concreteness, we will adopt λProlog notation. In λProlog, we could encode λ-terms using the following declarations:

```
kind exp type.
type app exp -> exp -> exp.
type lam (exp -> exp) -> exp.
```

The explicit type *string* and the expression constructor Var have disappeared entirely. In addition, the constructor lam for representing λ-expressions no longer takes a paired string and expression; instead, it takes a *meta-level function* mapping

[1]While the use of side-effects can be avoided by explicitly passing around a supply of fresh names, the resulting programs are more complex and no easier to reason about. This approach merely explicates the complexity implicit in the use of side-effects.

expressions to expressions. The encoding of λ-expressions in this syntax is as follows:

$$\begin{aligned}\ulcorner x \urcorner &= x \\ \ulcorner (e_1\ e_2) \urcorner &= \mathtt{app}\,\ulcorner e_1 \urcorner\,\ulcorner e_2 \urcorner \\ \ulcorner \lambda x.e \urcorner &= \mathtt{lam}\,(\lambda x.\ulcorner e \urcorner)\end{aligned}$$

where x denotes an object variable on the left side and a meta-variable on the right side, and $\lambda x.e$ denotes λ-abstraction in the meta-language on the right side.

In this encoding, the meta-language equates terms up to α-equivalence, as well as β and (sometimes) η-equivalence as defined by (1.8) and (1.9) in Table 1.1. Thus, it is not necessary to define α-equivalence explicitly for *exp*-terms; instead, the built-in equality of λProlog takes it into account. It is not necessary to define substitution explicitly either. Instead, to substitute e for x in e', we can simply evaluate the meta-level expression $(\lambda x.e')e$, and the built-in implementation of substitution will perform the desired calculation.

Another advantage of higher-order abstract syntax in λProlog is that relations such as typechecking can be implemented using λProlog's advanced logic programming features. In λProlog, program clauses are *hereditary Harrop formulas*, a generalization of Horn clauses in which goal formulas may include universal quantification and some forms of implication. These powerful features make it possible to implement relations such as λ-calculus typing as follows:

```
kind ty    type.
type arr   ty -> ty -> ty.
tc (lam (x\E x)) (arr T U) :- pi x \ (tc x T => tc (E x) U).
tc (app E1 E2) U :- tc E1 (arr T U), tc E2 T.
```

(where `pi x \ G` is λProlog syntax for $\forall x.G$). Note that there is no explicit representation of the context Γ, as shown in Table 1.1; instead, the logical context of λProlog itself is used to store the elements $x : \tau$ of Γ implicitly as hypotheses `tc x T`. The side-condition $x \notin FV(\Gamma)$ is enforced by the meta-level freshness constraint on $\forall$-bound variables in goals.

Higher-order abstract syntax encodings work very well for a large class of examples. For example, HOAS has been used very successfully to encode and reason about first-order logic, higher-order logic, and a variety of simple programming languages within logical frameworks such as λProlog and Twelf [104, 90, 105, 102].

Nevertheless, HOAS is not always possible or suitable for a problem. For example, if we wish to program with open terms, a HOAS encoding is not possible because free object-language variables would be translated to free meta-variables, which are not allowed in meta-language terms. A concrete instance of this problem is that it is not possible to deal with contexts or substitutions as an explicit data structure. In contrast, in FOAS, contexts and substitutions can be represented using lists of name-type or name-term pairs. Another drawback to HOAS is that it lacks a straightforward semantics, making reasoning about HOAS encodings using familiar mathematical techniques difficult [54]. In addition, higher-order unification, which is needed to execute λProlog programs, is undecidable [58]. As a result, HOAS programs can also be difficult to write, verify, and analyze.

1.3 Proposed Solution

1.3.1 Nominal Abstract Syntax

This dissertation explores an alternative way of programming with names, which we call *nominal abstract syntax*, in which names and name binding are encoded using built-in abstract data types called *name types* and *abstraction types*. Nominal abstract syntax is more complex than first-order abstract syntax, but considerably simpler than HOAS. Nominal abstract syntax provides just enough structure to capture α-equivalence while still admitting a simple first-order semantics and a decidable equality theory and unification problems. Nominal abstract syntax can be viewed as a formally sound reconstruction of intuitive reasoning on terms up to α-equivalence.

The underlying ideas of nominal abstract syntax were developed in the context of Fraenkel-Mostowski set theory (or FM-set theory), and used to develop a theory of binding syntax by Gabbay and Pitts [42, 43]. The key components of this approach are:

- a set $\mathbb{A}$ of infinitely many concrete *names* $\mathsf{a}, \mathsf{b}, \ldots$,
- a *swapping* operation $(\mathsf{a}\ \mathsf{b}) \cdot t$ that swaps two names within an expression,
- a *freshness* relation $\mathsf{a} \mathrel{\#} t$ that holds between a name and an expression when the latter is independent of the former,
- and an *abstraction* operation $\langle \mathsf{a} \rangle t$ that binds a name within an expression, and admits equality up to α-equivalence

In addition, Gabbay and Pitts identify two key principles:

- **Fresh name generation.** A name fresh for any expression (or for each of finitely many expressions) can always be found.
- **Equivariance.** Relations among expressions are invariant up to swapping; the choice of particular names is irrelevant.

It should be noted that the idea of identifying formulas up to one-to-one renamings of bound variables was already present in the work of Frege. It is only a small step from one-to-one renamings to name-swappings or permutations. On the other hand, nominal abstract syntax at present contains no built-in support for capture-avoiding substitution, but it is easy to define.

The key insight of Gabbay and Pitts' technique is that α-equivalence can be defined in terms of swapping and freshness. In particular, equality for abstractions can be axiomatized as follows:

$$\langle a \rangle x \approx \langle b \rangle y \iff (a \approx b \wedge x \approx y) \vee (a \# y \wedge x \approx (a\ b) \cdot y)\ .$$

Intuitively, if two abstractions are not literally equal, they may be equal if their bodies are equal up to swapping the bound names and the name bound on one side does not appear free on the other. It will be the task of much of this dissertation to construct a satisfactory semantic and logical foundation for this intuitive idea.

Nominal abstract syntax has been applied in developing the functional programming language FreshML by Shinwell, Pitts, and Gabbay [109, 116, 117]. FreshML permits users to define new abstract types of bindable names and includes a type construction for name binding. In addition, name-swapping, fresh name generation, abstractions, and pattern-matching against abstractions are built-in operations which support a cleaner form of programming with bindable names. The language implementation takes care of the details of actually generating names, performing routine renaming, and so on; moreover, it is free to do so in any way which respects the invariants of the name and abstraction types.

1.3.2 Nominal Logic Programming and αProlog

While the nominal abstract syntax approach is already being put into practical use in FreshML, the emphasis of FreshML is on making it easier to write programs dealing with names, not necessarily to reason about programs and definitions involving names. Specifications of programming languages and type systems are often presented as a collection of nondeterministic inference rules that do not correspond directly to function definitions in a functional programming language like FreshML.

Logic programming offers a compelling alternative to functional programming since it encourages programming in a declarative, relational style. Logic programs can be viewed as a collection of logical formulas defining a problem. Usually, the formulas are of a particularly convenient form: they are *Horn clauses* of the form $A :- G_1, \ldots, G_n$, where A, $G_1, \ldots, G_n$ are atomic formulas; such a clause is read as "A is true provided formulas $G_1, \ldots, G_n$ are true". Horn clause logic programs can be run by interpreting logical connectives as proof-search operations, using backchaining and depth-first search. This is the approach taken in classical logic programming (Prolog) and in this dissertation.

The subject of this dissertation is *nominal logic programming*, that is, logic programming with nominal abstract syntax. I present a nominal logic programming language called αProlog ("alpha-Prolog") [17, 18]. Like FreshML, αProlog includes built-in support for names and name binding, and equates expressions up to α-equivalence. However, αProlog supports a distinctly different style of programming than FreshML. "Paper" specifications of common programming languages and logics can be transcribed very directly into αProlog program clauses.

This sets αProlog apart from all other tools for either programming with or reasoning about languages with names.

For example, the language of λ-terms can be encoded using the following αProlog type declarations:

$$\begin{array}{c} id : \mathbf{name\_type} \qquad exp : \mathbf{type} \qquad ty : \mathbf{type} \\ var : id \to exp \quad app : exp \times exp \to exp \quad lam : \langle id \rangle exp \to exp \quad arr : (ty, ty) \to ty \end{array}$$

Since abstractions are considered equal up to "safe" renaming, the built-in equality on closed *exp* terms coincides with α-equivalence of (possibly non-ground) λ-terms. Therefore, equality and unification take α-equivalence into account "for free" in this encoding, and no added effort from the programmer is needed, just as in higher-order abstract syntax.

Unlike higher-order abstract syntax, however, meta-level λ-abstraction and application are not built-in, and unification does not take substitution, β-equivalence, and η-equivalence into account. Therefore, unification remains essentially first-order and decidable.

The definition of substitution in αProlog is strikingly similar to the informal version defined using the Barendregt Variable Convention (shown in Table 1.1):

$$\begin{array}{lcl} \mathbf{func}\ subst(exp, exp, id) = exp. & & \\ subst(var(\mathsf{x}), T, \mathsf{x}) & = & T. \\ subst(var(\mathsf{y}), T, \mathsf{x}) & = & var(\mathsf{y}). \\ subst(app(E_1, E_2), T, \mathsf{x}) & = & app(subst(E_1, T, \mathsf{x}), subst(E_2, T, \mathsf{x})). \\ subst(lam(\langle \mathsf{y} \rangle E_1), T, \mathsf{x}) & = & lam(subst(E_1, T, \mathsf{x})) :- \mathsf{y} \# T. \end{array}$$

Here, x and y are different name constants, so are not equal as terms. This encodes the side-condition $x \neq y$ in the second and fourth clauses. Note that in the fourth clause, there is a side-condition that x must be "fresh" for T, corresponding to the side-condition $x \notin FV(T)$ in the classical definition. In αProlog, the abstracted name x may be assumed to be fresh without loss of generality, so *subst* still defines a (total) function. Therefore, additional clauses such as

$$subst(lam(\langle \mathsf{x} \rangle E_1, T, \mathsf{x}) = lam(\langle \mathsf{x} \rangle E_1)$$

while still correct, are redundant, i.e. derivable from the other clauses.

Similarly, the typing judgment $\Gamma \vdash e : \tau$ can be implemented using the following Horn clauses:

$$\begin{array}{lcl} \mathbf{pred}\ tc([(id, ty)], exp, ty). & & \\ tc(G, var(X), T) & :- & mem((X, T), G). \\ tc(G, app(M, N), T') & :- & tc(G, M, arr(T, T')), tc(G, N, T). \\ tc(G, lam(\langle \mathsf{x} \rangle M), arr(T, T')) & :- & \mathsf{x} \# G, tc([(\mathsf{x}, T) | G], M, T'). \end{array}$$

Here, $mem(A, L)$ is the list-membership predicate, which succeeds if A is a member of L. Again, the freshness constraint $\mathsf{x} \# G$ corresponds to the side-condition

$x \notin FV(\Gamma)$. Thus in both of these instances we can write relations and functions as we might do in an informal presentation assuming the Barendregt Variable Convention. But there is no informality here: the meanings of *subst* and *tc* are well-defined functions and relations with the properties we would expect them to have in a completely formal treatment. In fact, these αProlog programs themselves constitute formal definitions of substitution, typechecking, etc. which can be used to reason about the λ-calculus.

1.4 Outline

My thesis is that

> Nominal logic programming is a powerful technique for programming with names and binding.

To support this thesis, I shall present the design of αProlog, a particular nominal logic programming language; give examples of programs that can be written more easily in αProlog than any other language; provide a semantics of nominal logic programs; and investigate the complexity of and algorithms for nominal constraint solving problems. Along the way, many technical issues arise and I shall show how to resolve them.

The first substantial technical problem that is solved in this dissertation is the development of a logic that is a suitable foundation for nominal logic programming. Just as classical Prolog logic programming is based on first-order logic, we require a logic upon which to base αProlog. An obvious candidate is *nominal logic* (NL), a variant of first-order logic defined by Pitts which includes names, swapping, freshness, abstraction, and appropriate axioms [108]. However, Pitts' nominal logic has several drawbacks when considered as a foundation for nominal logic programming. First, it is presented using a Hilbert-style axiom system, whereas Gentzen-style inference rule systems are often better suited for understanding logic programming. Second, NL is incomplete with respect to Pitts' finite-support semantics: that is, there exist consistent theories of NL with no finite-support models. Third, any language of NL containing name-constants (or closed terms) is inconsistent. Therefore, Herbrand's Theorem, an important step in understanding the semantics of logic programming, trivially fails in NL because not enough closed terms exist.

The second technical problem I address is the development of a sensible semantics for nominal logic programming. Because of the presence of freshness constraints in αProlog programs, the semantic framework of *constraint logic programming* (CLP) [61] is well-suited to defining the semantics of αProlog. In this framework, the high-level logical aspects of the semantics are separated cleanly from the low-level constraint-solving problems that need to be solved during execution. Constraint logic programming is a very general framework for analyzing first-order logic programming with constraints; unfortunately, however, it cannot be used to analyze nominal logic programming. As a result, it is necessary to provide new definitions and prove new theorems concerning the semantics of nominal logic programs. This development follows existing work on the semantics of CLP [62].

The third technical problem addressed in this dissertation is that of solving the constraints arising during the execution of nominal logic programs. This problem is analogous to the unification problems arising in first-order, higher-order, and other variants of logic programming; however, in nominal logic programming, both equality, freshness, and *atomic logical equivalence* constraints must be solved. These problems are more general than the nominal unification problem solved by Urban, Pitts, and Gabbay's algorithm [126, 127], and so it is necessary to investigate the complexity of these problems and develop new algorithms for solving them.

The structure of the rest of this dissertation is as follows.

Chapter 2 presents the αProlog language, describes its type system, unification procedure, and execution algorithm informally, and gives several example programs demonstrating its expressive power.

In Chapter 3, I introduce a revised theory of nominal sets which may include elements with infinite but "small" support. This theory is subsequently used for both the syntax and semantics of nominal logic.

In Chapter 4, I define a new version of nominal logic, which includes name-constants and is equipped with a Gentzen-style proof system admitting cut-elimination. The semantics of nominal logic is given using nominal sets.

In Chapter 5, it is shown that nominal logic is sound and complete with respect to the revised semantics. In addition, an appropriate version of Herbrand's Theorem is proved.

Chapter 6 presents a CLP-style operational and denotational semantics for αProlog, and shows them to be equivalent by proving the relevant soundness and completeness properties. In addition, this chapter identifies the *nominal constraints* which need to be solved during the execution of nominal logic programs.

Chapter 7 discusses the nominal constraint solving problems arising in αProlog. First, it is shown that all of the problems are **NP**-hard in general, and most of them are **NP**-complete. (The worst-case complexity of one problem is left open). Next, sound and complete algorithms for solving the constraints are developed. The chapter concludes with a discussion of the shortcomings of these algorithms, some efficient special cases, and directions for future work on nominal constraint solving.

Chapter 8 compares αProlog with other programming languages that support advanced programming with names and binding, and discusses other work that is related to nominal logic programming.

Chapter 9 reviews the results presented in this dissertation, discusses future research directions, and concludes.

Chapter 2

The αProlog Language and Examples

Inside every large program is a small program struggling to get out.

—*C. A. R. Hoare*

This chapter presents the logic programming language αProlog. We begin with an overview of the core language. We present the syntax and type system of αProlog. Next follow several examples of αProlog programs.

2.1 Language Overview

2.1.1 Syntax

The terms of αProlog are called *nominal terms* t, constructed according to the syntax in Table 2.1, where X, Y are *(logic) variables*, f, g are *function symbols*, p, q are *relation symbols*, and a, b are *name-constants.* Note that name-constants, variables, and function symbols are distinct syntactic classes. We use sans-serif letters $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{x}, \mathsf{y}, \mathsf{z}$ for name-constants, and uppercase letters to denote variables; other identifiers are assumed to be function or relation symbols, according to context. In addition, function symbols are subdivided into two classes: *defined function symbols* and *constructors.*

Table 2.1: αProlog syntax: terms and formulas

Terms	t	$::=$	$X \mid f(\vec{t}) \mid \langle \mathsf{a} \rangle t \mid (\mathsf{a}\ \mathsf{b})t$
Constraints	C	$::=$	$\mathsf{a} \# t \mid t \approx u$
Atomic formulas	A	$::=$	$p(\vec{t})$
Goals	G	$::=$	$C \mid A \mid \top \mid G_1, G_2 \mid G_1; G_2$
Program clauses	P	$::=$	$A :- G \mid f(\vec{t}) = t :- G$

Terms of the form $\langle \mathsf{a} \rangle t$ are called *abstractions*, and terms of the form $(\mathsf{a}\ \mathsf{b})t$ are called *swappings*. Also, note that variables cannot be used to form abstractions and swappings, i.e., $\langle X \rangle t$ and $(X\ Y)t$ are not well-formed terms.

The notation for swappings may be confusing at first glance. It is based on standard notation in group theory, where $(x\ y)$ denotes a permutation that exchanges the values of x and y, leaving all other elements fixed. Intuitively, $(a\ b)t$ is the result of *swapping* two name-constants a and b everywhere within a term t. Thus, $(\mathsf{a}\ \mathsf{b})f(\mathsf{a},\mathsf{b},\mathsf{c}) \approx f(\mathsf{b},\mathsf{a},\mathsf{c})$. Also, the abstraction function symbol denotes *object-level* name-abstraction within the term language. This should not be confused with ordinary variable binding: for example $\langle \mathsf{a} \rangle \mathsf{a}$ and $\langle \mathsf{b} \rangle \mathsf{b}$ are *syntactically different* terms which denote the *same value*, whereas $\langle \mathsf{a} \rangle X$ and $\langle \mathsf{b} \rangle X$ may or may not denote the same value, depending on X.

Atomic formulas (or atoms) A are formulas of the form $p(t_1, \ldots, t_n)$. Constraints C are equations $t \approx u$ or freshness constraints $\mathsf{a} \# u$. Note that the left-hand side of $\#$ must be a name-constant. *Goals* (or *queries*) G are constructed using the grammar rules in Table 2.1, and include conjunction (G, G'), disjunction $(G; G')$, logical truth $(\top)$, atomic predicates, and constraints. Program clauses include Horn clauses of the form $A :- G$ and function-definition clauses of the form $f(\vec{t}) = u :- G$. The latter permit the definition of functional relations; this is a standard extension to logic programming. We abbreviate clauses $A :- \top$ and $f(t) = u :- \top$ as simply A and $f(t) = u$.

Term and propositional constants c, p are taken by convention to be the special case of a function or relation symbol applied to the empty list of arguments; we write c for $c()$ and p for $p()$.

Example 2.1.1. The following are well-formed program clauses:

$$p(X) :- q(X, X), X \approx Y. \quad f(\langle \mathsf{a} \rangle X, Y) = X :- q((\mathsf{a}\ \mathsf{b})X), \mathsf{a} \# Y.$$

The formulas $p((Y\ Z)X)$, $p(\langle Y \rangle Z)$ and $Y \# X$ are ill-formed because of the restriction that variables (here Y and Z) may not appear in place of names in swappings, abstractions, and freshness constraints.

2.1.2 Type System

Traditionally, Prolog-like languages are untyped. However, in programming practice, it is usually an advantage to be able to check that the arguments to functions or predicates are sensible, to rule out nonsense programs that try to add a list to a number or concatenate a boolean value to a string. In dynamically typed languages such as Prolog and Lisp, such nonsense programs result in run-time failure, and programmers are encouraged to use built-in predicates to write "bulletproofing" code that handles type errors more gracefully. Programming languages such as SML, Haskell, Mercury, and λProlog are based on *static type systems* that rule out such ill-formed programs prior to execution. Such languages also include *polymorphism*; that is, types can contain variables that can be instantiated during

Table 2.2: αProlog syntax: kinds, types, and declarations

Kinds	κ	$::=$	$\mathbf{type} \mid \mathbf{name\_type}$
Types	σ, ν, τ	$::=$	$\alpha \mid tcon(\sigma_1, \ldots, \sigma_n) \mid \langle\nu\rangle\sigma$
Declarations	D	$::=$	$tcon : (\kappa_1, \ldots, \kappa_n) \to \kappa \mid f : (\sigma_1, \ldots, \sigma_n) \to \tau$
		$\mid$	$\mathbf{pred}\ p(\sigma_1, \ldots, \sigma_n) \mid \mathbf{func}\ f(\sigma_1, \ldots, \sigma_n) = \sigma$
		$\mid$	$\mathbf{type}\ \tau = \sigma$

typechecking, so that, for example, list-manipulation functions need be written only once rather than repeatedly for each type of interest.

Static typing confers a number of advantages: types serve as lightweight program specifications and program errors are often caught at compile-time rather than causing bugs at run-time which must be tracked down and fixed. And types can often be inferred by the language implementation, saving the programmer from having to write down explicit types.

αProlog is a statically-typed, polymorphic language. Its type system permits user-defined *algebraic datatypes* and allows polymorphic function and predicate definitions. These features are standard in strongly-typed functional languages such as ML, Haskell, and strongly-typed logic languages such as λProlog and Mercury. In fact, αProlog's type system is almost a special case of those of the latter two languages: the only difference is the inclusion of name types and abstractions. The syntax of types and type declarations is summarized in Table 2.2.

In αProlog, types are classified into two basic *kinds*: **type** and **name_type**. Every type is of kind **type**, whereas only name-types are of kind **name_type**. Name types are types inhabited only by name-constants. The programmer may declare new types and name-types as follows:

$$exp : \mathbf{type}. \qquad id : \mathbf{name\_type}.$$

In fact, there are no built-in name-types, so all name-types must be introduced in this way.

Also, αProlog allows type-level functions, or *type constructors*, which take several types as arguments and produce a type. For example, *list* takes a type σ and produces a type *list* σ (abbreviated $[\sigma]$) consisting of lists of values from σ; we write $list : \mathbf{type} \to \mathbf{type}$ to indicate that *list* constructs types from types. The abstraction constructor $\langle\cdot\rangle\cdot$ takes a name-type ν and a type σ and produces a type $\langle\nu\rangle\sigma$ inhabited by abstractions. Thus, we write $abs : (\mathbf{name\_type}, \mathbf{type}) \to \mathbf{type}$. In general, a type constructor c with signature $(\bar{\kappa}) \to \mathbf{type}$ takes a list of types of kinds $\kappa_1, \ldots, \kappa_n$ and produces a type of kind κ. Type constructors are allowed to construct name types.

Similarly, αProlog terms are classified by types. Each constant or variable has an associated type; a constructor has an associated function type $(\bar{\sigma}) \to \tau$ indicat-

Table 2.3: Built-in types, constructors, and constants

Type	Constructors
$\mathit{bool} : \mathbf{type}$	$\mathit{true}, \mathit{false}$
$\mathit{int} : \mathbf{type}$	$1, 2, 3 \ldots$
$\mathit{char} : \mathbf{type}$	`'a'`, `'b'`, ...
$[\cdot] = \mathit{list} : \mathbf{type} \to \mathbf{type}$	$[] : [A], [\cdot\|\cdot] : (A, [A]) \to A$
$\mathit{string} = [\mathit{char}] : \mathbf{type}$	`"abcd"` = [`'a'`, `'b'`, `'c'`, `'d'`]...
$\mathit{opt} : \mathbf{type} \to \mathbf{type}$	$\mathit{none} : \mathit{opt}\ A, \mathit{some} : A \to \mathit{opt}\ A$
$\langle\cdot\rangle\cdot = \mathit{abs} : (\mathbf{name\_type}, \mathbf{type}) \to \mathbf{type}$	$\langle\cdot\rangle\cdot : (A, T) \to \langle A\rangle T$

ing that the function expects arguments of type $\sigma_1, \ldots, \sigma_n$ and produces a result of type τ. The result type τ must be a user-defined type. Defined function and relation symbols are associated with types using the **pred** and **func** declarations. Type abbreviations may be defined using the **type** *con* $\vec{\alpha} = \sigma$ declaration. Types may contain variables; type variables in constructor, **pred**, **func**, and **type** declarations are interpreted as implicitly universally quantified. Thus, for example, $\mathbf{func}([A]) = A$ is the type of a function that takes a list of elements of some type A and returns a value of the same type.

There are several standard built-in types, summarized in Table 2.3. All of them are standard except for the abstraction type constructor *abs*. Note that strings are defined as lists of characters, with string constants as syntactic sugar for list notation. We also may view the abstraction type and term constructors as ordinary type and term constructors for the purpose of typing, and we can give the swapping operation the polymorphic type

$$(\cdot\ \cdot)\cdot : \forall A : \mathbf{name\_type}, T : \mathbf{type}.(A, A, T) \to T$$

In addition, we may view formulas as having a type **prop**. Then we can give the built-in formulas the following polymorphic types:

$$\begin{array}{rcl} \cdot \# \cdot & : & \forall A : \mathbf{name\_type}, T : \mathbf{type}.(A, T) \to \mathbf{prop} \\ \cdot \approx \cdot & : & \forall T : \mathbf{type}.(T, T) \to \mathbf{prop} \end{array}$$

Why do we introduce two different types, *bool* and **prop**, for (arguably) the same thing? Intuitively, *bool* is the type consisting of two truth-values true and false; these truth values can be compared and manipulated using Boolean operations. In contrast, **prop** consists of formulas which might have free (logical) variables, and it often does not make sense to perform arbitrary Boolean manipulations on **prop**, or to compare two propositions for equality. In other words, **prop** consists of computations, whereas *bool* consists of truth values.

Parametricity

The type system of αProlog is intended to be parametric, in order to avoid the need for typechecking at run time. This means that the behavior of polymorphic predicates and functions must be independent of the types of their arguments. For example, the following program is illegal. (The first line is a type declaration and the next two lines are program clauses defining the meaning of *dyn*.)

$$\begin{array}{l}\textbf{pred}\ dyn(A, string).\\ dyn(0, \texttt{"integer"}).\\ dyn(true, \texttt{"bool"}).\end{array}$$

In contrast, some typed logic programming languages, such as λProlog, employ non-parametric (or intensional) polymorphism and dynamic typing. A program like the above would be legal in λProlog, and in addition λProlog includes a type-annotation term constructor $t : \sigma$ that can be used to test types at run-time. Though more powerful, intensional polymorphism requires type information to be maintained throughout execution so that typechecking can be performed at run-time; in contrast, parametric polymorphism permits most or all type information to be erased after type-checking.

2.1.3 How Programs Run

In Prolog (and most other logic programming languages), we view a program as a set of clauses together with a *goal* or *query* which we wish to solve. When we wish to solve a goal A, we search the program for a clause of the form $A :- G_1, \ldots, G_n$, and replace the goal A with the subgoals $G_1, \ldots, G_n$. Then we proceed recursively to solve all the new subgoals (as well as any other outstanding subgoals that might need to be solved in addition to A). This is called *backchaining*. If there is no matching clause, then Prolog *backtracks* to a point where an alternative backchaining step could be taken.

This proof search procedure is a bit naïve, since it assumes that whenever A can be proved from a program, there exists a clause $A :- G$ whose head is literally A. This is true if we limit programs to contain ground propositions only (that is, atomic predicates contain no variables.) However, as soon as variables enter the picture, this assumption is no longer true, for we may have a goal $p(c)$ and a clause $p(X) :- q(X, d)$ which "matches" $p(c)$ without being syntactically equal. Instead, they are equal modulo a *unifying substitution*, in this case $p(c) \approx p(X)$ when $X = c$, so we may solve $p(c)$ if we can solve $q(c, d)$. Fortunately, there are efficient algorithms for *unification*, the problem of determining whether two terms can be made equal by applying a substitution. Therefore, in Prolog, execution proceeds by recursively solving goals using unification to determine when to apply a clause. This proof search procedure combining backchaining and unification is sometimes called *SLD-resolution*.

αProlog programs are executed in a fashion very similar to Prolog programs. But in αProlog, there two complications. The first is the fact that the equality theory of αProlog is not simply syntactic equality, but a coarser equivalence relation which identifies terms up to a form of α-equivalence. In particular, two abstractions such as $\langle \mathsf{a} \rangle \mathsf{a}$ and $\langle \mathsf{b} \rangle \mathsf{b}$ are considered equal, so if ordinary first-order unification were used, correct answers would be missed. Therefore, in αProlog, it is at least necessary to generalize first-order backchaining and unification to unify up to equality in nominal logic, or *nominal unification.* We will show in Chapter 7 that nominal unification can be hard in general. Fortunately, an efficient algorithm for the special case of nominal unification needed in αProlog has been developed by Urban, Pitts, and Gabbay [126, 127]: as long as the names a, b in abstractions $\langle \mathsf{a} \rangle t$ and swappings $(\mathsf{a}\ \mathsf{b})t$ are ground (that is, constants), their algorithm produces unique most general unifiers.

Here is an example. Suppose the program contains a clause such as

$$p(\langle \mathsf{a} \rangle X) :- q([\mathsf{a}], X).$$

and we wish to solve the goal $p(\langle \mathsf{b} \rangle \mathsf{b})$. The (unique) nominal unifier (i.e., module α-equivalence) of $p(\langle \mathsf{a} \rangle X)$ and $p(\langle \mathsf{b} \rangle \mathsf{b})$ is $X = \mathsf{a}$, so we may proceed by solving the subgoal $q([\mathsf{a}], \mathsf{a})$.

If both the goal and clause head contain variables then the situation can be more complex. Consider the program clause head $p(Z, Z)$ and the goal $p(\langle \mathsf{b} \rangle Y, \langle \mathsf{a} \rangle X)$. Then the most general nominal unifier is $X = (\mathsf{a}\ \mathsf{b})Y$ with the side-constraint $\mathsf{a} \# Y$. Such freshness side-conditions are necessary to guarantee that α-equivalence is not violated. If the side-condition were ignored, then we could obtain an incorrect solution such as $Y = a$. This solution is incorrect because $\langle \mathsf{a} \rangle \mathsf{b} \not\approx \langle \mathsf{b} \rangle \mathsf{a}$, that is, the terms are not α-equivalent. There is a seeming asymmetry in the reduction to the constraint $\mathsf{a} \# Y$: why do we not need to check $\mathsf{b} \# X$ as well? In fact, if $X = (\mathsf{a}\ \mathsf{b})Y$ and $\mathsf{a} \# Y$, then it follows that $\mathsf{b} \# X$.

As an important aside, note that in Prolog, in order to obtain completeness (not miss any answers) it is necessary to *freshen* all the variables in a clause before unifying. This ensures that there are no false dependences between the goal variables and those in the clause. For example, the clause $p(X, c)$ does not unify with the goal $p(d, X)$, but the freshened clause $p(X', c)$ does, producing the answer $X = c, X' = d$. In αProlog, we treat names similarly to variables, in that names are freshened whenever an attempt is made to unify a clause head with a goal. Since we have so far treated name-constants as ordinary constants, this may not seem appropriate. In fact, in nominal logic, name-constants behave like variables in some respects; in particular, they can be bound by a "fresh name" quantifier И. It is our logical treatment of this quantifier that will justify the freshening of name-constants during backchaining. This approach is proved both sound and complete in Chapter 6.

The second source of complication in the execution of αProlog programs arises from the *equivariance* property of nominal logic, which asserts that if some atomic formula holds, then any variant obtained by swapping names also holds. For example, if $p(\mathsf{a})$ is true, then so is $p(\mathsf{b})$, $p(\mathsf{a}')$, ... Similarly, if $q(\mathsf{a}, \mathsf{b})$ holds then so does $q(\mathsf{a}', \mathsf{b}')$ for any *distinct* a', b' (but not necessarily for $q(\mathsf{a}', \mathsf{a}')$, etc.) This may seem arbitrary, but it is actually a cornerstone of nominal logic and cannot be ignored. As a more concrete motivating example, equivariance tells us that a relation such as typechecking is *a priori* closed under renaming, thus, from

$$typ([], lam(\langle \mathsf{x} \rangle lam(\langle \mathsf{y} \rangle var(\mathsf{x}))), T \to U \to T)$$

we may immediately infer

$$typ([], lam(\langle \mathsf{z} \rangle lam(\langle \mathsf{w} \rangle var(\mathsf{z}))), T \to U \to T)$$

(assuming no names occur in T or U).

Equivariance is convenient, powerful, and necessary from a semantic point of view (as we shall see in Chapter 4), but complicates the implementation of αProlog. Specifically, because of equivariance, backchaining based on nominal unification, while sound and less *in*complete than ordinary first-order backchaining, is not complete. Thus, there may be consequences of an αProlog program (viewed as a nominal theory) that cannot be computed using nominal backchaining. For example, if the program clause is

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

then an attempt to solve the goal $p(\mathsf{a})$ will fail, because $p(\mathsf{a})$ will be freshened to $p(\mathsf{a}')$ before we attempt to unify $p(\mathsf{a})$ and $p(\mathsf{a}')$; this attempt will fail because $\mathsf{a} \not\approx \mathsf{a}'$.

Instead, it is necessary to consider an even coarser form of equivalence that equates $p(\vec{t})$ and $p((\mathsf{a}\ \mathsf{b})\vec{t})$. Unfortunately, even for the restricted terms used in nominal unification, deciding unifiability up to this equivalence (a problem we call *equivariant unification*) is **NP**-hard.

This limitation can frequently be circumvented, but usually only by avoiding the kind of open-term manipulations that distinguish nominal abstract syntax from other approaches. In fact, as we shall discuss briefly in Chapter 7, there is a large class of interesting αProlog programs that behave correctly even if only nominal unification is used.

In this chapter we will write programs that rely on equivariant unification to run properly; however, at present these programs cannot be executed very efficiently. There are two reasons for doing this: first, although in many cases the programs could also be written so as to avoid reliance on equivariant unification, the resulting programs would be considerably more complex and less clear; and second, these programs help to establish the potential usefulness and desirability of practical equivariant unification.

Many unification problems have practically useful algorithms for common cases despite being **NP**-complete in general. We hope this will turn out to be the case for equivariant unification (and generalized nominal unification) as well. However, the development of such techniques is beyond the scope of this dissertation and is an extremely important area for future work.

We now give two small examples of αProlog programs.

Example 2.1.2. Consider the following program clause:

$$p(\langle \mathsf{a} \rangle X, \langle \mathsf{a} \rangle Y) \mathrel{:-} q(X, Y).$$

This clause matches $p(t, u)$ provided both t, u are abstractions with the same name at the head. In fact, any two abstractions can be renamed to a *fresh* name a, so αProlog's execution strategy of freshening names guarantees that the above program clause will match any abstractions t, u.

For example, consider $p(\langle \mathsf{a} \rangle f(\mathsf{a}, \mathsf{b}), \langle \mathsf{b} \rangle f(\mathsf{a}, \mathsf{b}))$. Unifying against a fresh instance of $p(\langle \mathsf{a}' \rangle X, \langle \mathsf{a}' \rangle Y)$ yields the unifier $X = f(\mathsf{a}', \mathsf{b}), Y = f(\mathsf{a}, \mathsf{a}')$. If a had not been freshened to a', however, unification would fail because $\langle \mathsf{b} \rangle f(\mathsf{a}, \mathsf{b})$ is not α-equivalent to $\langle \mathsf{a} \rangle X$ for any instantiation of X.

Example 2.1.3. Consider the program *dunion* defined as follows:

$$\begin{array}{ll} \textbf{pred}\ \mathit{dunion}([A], [A], [A]). & \\ \mathit{dunion}([], L, L). & \\ \mathit{dunion}([\mathsf{a}|L], M, [\mathsf{a}|N]) & \mathrel{:-}\ \mathsf{a} \# M, \mathit{dunion}(L, M, N). \end{array}$$

This predicate takes two lists of names and merges them, checking that no names in the first list are mentioned in the second. If the two input lists contain no duplicates, then neither does the output. Thus, if lists are used to represent sets, *dunion* calculates the disjoint union of sets, failing if it does not exist.

2.1.4 Standard Logic Programming Extensions

There are several features found in Prolog and other logic programming languages which we will use in example programs but not deal with formally in this dissertation. They include function definitions (which we have already encountered), infix operators, disjunctive goals, the "cut" proof-search operator, negation as failure, if-then-else goals, and definite clause grammars. In this section we briefly explain these features in case they are unfamiliar.

Function definitions are often incorporated into logic programming by unfolding or *flattening* the definitions. This means replacing the defined function symbol with a relation symbol such that $p(\vec{x}, y)$ if and only if $f(\vec{x}) = y$. This transformation is performed by many Prolog implementations, although more advanced techniques are known for combining functional programming and logic programming [50].

Most programming languages include infix operators for arithmetic and boolean operations. This is the case in Prolog and αProlog as well. In addition, Prolog supports user-defined infix operators, which has become a standard feature in logic and functional languages. αProlog provides limited support for infix operators as well.

Disjunctive goals $G; G'$ are executed by attempting to solve G first, then G' if no solution to G is found. Programs containing such goals can be translated to pure Horn clause programs by, for example, replacing $A :- G, (G_1; G_2), G'$ with $A :- G, P, G'$ using an auxiliary predicate P defined by two clauses $P :- G_1$, $P :- G_2$.

The cut operator ! is a nonlogical construct which discards the backtracking state associated with the current goal when it is executed. It can be used to commit to a particular solution to a goal, which can increase the efficiency of a program by avoiding unnecessary backtracking. However, cut may damage the declarative transparency of a program. For example, the program

$$p(X) :- q(X), !, r(X). \quad q(a). \quad q(b). \quad r(X).$$

produces only the answer $X = a$ to the query $p(X)$, because the cut "commits" to the first answer $q(a)$ found for $q(X)$, and the second answer $X = b$ is missed. However, the more explicit query $p(b)$ succeeds, because the search for a solution $q(b)$ succeeds before the cut is processed. We will not actually use cut in the examples, so will not discuss it further.

Negation as failure means that goals of the form $not(G)$ are permitted. Such a goal succeeds if the search for a proof of G terminates in failure, and fails otherwise. Goals for which proof search terminates with failure are said to *finitely fail*. *If-then-else goals* are goals of the form $G \to G_t | G_f$. If G finitely fails, then G_f is executed, and otherwise G_t is executed. In addition, if backtracking occurs during the execution of G_t, then alternative solutions to G are not considered (similar to cut). Negation-as-failure and if-then-else are logically better behaved than cut, but can lead to strange behavior as well.

Definite clause grammars [101] are a standard extension to logic programming languages. It is straightforward to incorporate DCGs into αProlog. They are not used in any of the examples, so we will not describe them in any more detail; instead see for example [21].

2.2 Examples

2.2.1 The λ-calculus

Recall that the λ-calculus includes terms of the form

$$t ::= x \mid t\ t' \mid \lambda x.t$$

where x is a variable name and t, t' are terms. We encode this syntax with the αProlog declarations

$$\begin{array}{lcl@{\qquad}lcl} exp & : & \mathbf{type}. & id & : & \mathbf{name\_type}. \\ var & : & id \to exp. & app & : & (exp, exp) \to exp. \\ lam & : & \langle id \rangle exp \to exp. & & & \end{array}$$

Example 2.2.1 (Typechecking and inference). First, we consider the problem of typechecking λ-terms. The syntax of types includes type variables α and function types $\tau \to \tau'$ constructed from types τ and τ', and can be encoded as follows:

$$\begin{array}{lclllcl} tid & : & \textbf{name\_type}. & ty & : & \textbf{type}. \\ tvar & : & tid \to ty. & arr & : & (ty, ty) \to ty. \end{array}$$

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

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{type}\ ctx = [(id, ty)].} \\ \multicolumn{3}{l}{\textbf{pred}\ typ(ctx, exp, ty).} \\ typ(G, var(X), T) & :- & mem((X,T), G). \\ typ(G, app(M,N), T') & :- & typ(G, M, arr(T,T')), typ(G, N, T). \\ typ(G, lam(\langle \mathsf{x} \rangle M), arr(T,T')) & :- & \mathsf{x} \# G, typ([(\mathsf{x}, T)|G], M, T'). \end{array}$$

The predicate mem is the usual predicate for testing list membership. The side-condition $x \notin Dom(\Gamma)$ is translated to the freshness constraint $\mathsf{x} \# G$.

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

$$typ(G_1, lam(\langle \mathsf{x}_1 \rangle M_1), arr(T_1, U_1)) :- \mathsf{x}_1 \# G_1, typ([(\mathsf{x}_1, T_1)|G_1], M_1, U_1)$$

which unifies with the goal with $[G_1 = [], M_1 = lam(\langle \mathsf{y} \rangle var(\mathsf{x}_1)), T = arr(T_1, U_1)]$. This yields subgoal $\mathsf{x}_1 \# [], typ([(\mathsf{x}_1, T_1)|G_1], M_1, U_1)$. The first conjunct is trivially valid since G_1 is a constant. The second is solved by backchaining against the third typ-rule again, producing unifier $[G_2 = [(\mathsf{x}_1, T_1)], M_2 = var(\mathsf{x}_1), U_1 = arr(T_2, U_2)]$ and subgoal $\mathsf{x}_2 \# [(\mathsf{x}_1, T_1)], typ([(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], var(\mathsf{x}_1), U_2)$. The freshness subgoal reduces to the constraint $\mathsf{x}_2 \# T_1$, and the typ subgoal can be solved by backchaining against

$$typ(G_3, var(X_3), T_3) :- mem((X_3, T_3), G_3)$$

using unifier $[G_3 = [(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], X_3 = \mathsf{x}_1, T_3 = U_2]$. 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 $\lambda x.\lambda y.x$.

Example 2.2.2 (Substitution and Normalization). As mentioned in Chapter 1, in αProlog the substitution function can be written as follows:

$$\begin{array}{lcl} \textbf{func}\ subst(exp, exp, id) & = & exp. \\ subst(var(\mathsf{x}), P, \mathsf{x}) & = & P. \\ subst(var(\mathsf{y}), P, \mathsf{x}) & = & var(\mathsf{y}). \\ subst(app(M,N), P, \mathsf{x}) & = & app(subst(M, P, \mathsf{x}), subst(N, P, \mathsf{x})). \\ subst(lam(\langle \mathsf{y} \rangle M), P, \mathsf{x}) & = & lam(\langle \mathsf{y} \rangle subst(M, P, \mathsf{x})) :- \mathsf{y} \# P. \end{array}$$

This program makes maximum use of names: for example, in the second clause we use distinct names x and y to distinguish from the first clause, and the abstracted name in the fourth clause is also assumed fresh for P.

Next we define evaluation for λ-terms. We first define a relation which expresses β-reduction:

$$\begin{array}{l}
\textbf{pred}\ \mathit{beta}(\mathit{exp}, \mathit{exp}).\\
\mathit{beta}(\mathit{app}(\mathit{lam}(\langle \mathsf{x}\rangle M), M'), N) \quad :- \quad N = \mathit{subst}(M, M', \mathsf{x}).
\end{array}$$

Now we define a relation *step* that express one-step reduction of λ-terms.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{pred}\ \mathit{step}(\mathit{exp}, \mathit{exp}).}\\
\mathit{step}(M, M') & :- & \mathit{beta}(M, M').\\
\mathit{step}(\mathit{app}(M, N), \mathit{app}(M', N)) & :- & \mathit{step}(M, M').\\
\mathit{step}(\mathit{app}(M, N), \mathit{app}(M, N')) & :- & \mathit{step}(N, N').\\
\mathit{step}(\mathit{lam}(\langle \mathsf{x}\rangle M), \mathit{lam}(\langle \mathsf{x}\rangle M')) & :- & \mathit{step}(M, M').
\end{array}$$

Using if-then-else, we can express the normalization relation, that associates a normal form to a λ-expression, if it exists.

$$\begin{array}{l}
\textbf{pred}\ \mathit{nf}(\mathit{exp}, \mathit{exp}).\\
\mathit{nf}(M, M') :- \mathit{step}(M, M'') \rightarrow \mathit{nf}(M'', M') | M = M'.
\end{array}$$

This says that if M can take a step to M'', then M's normal form is the same as the normal form of M'', whereas if M cannot take a step then M is already in normal form.

Example 2.2.3 (α-inequivalence). Consider the problem of testing whether two λ-terms are *not* equivalent up to α-renaming. Using negation-as-failure, such a program is trivial:

$$\mathit{neq}(M, M') :- \mathit{not}(M \approx M').$$

However, this program is easy to write in αProlog, even without negation-as-failure:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{neq}(\mathit{var}(\mathsf{x}), \mathit{var}(\mathsf{y})).}\\
\mathit{neq}(\mathit{app}(M, N), \mathit{app}(M', N')) & :- & \mathit{neq}(M, M'); \mathit{neq}(N, N').\\
\mathit{neq}(\mathit{lam}(\langle \mathsf{x}\rangle M), \mathit{lam}(\langle \mathsf{x}\rangle M')) & :- & \mathit{neq}(M, M').\\
\multicolumn{3}{l}{\mathit{neq}(\mathit{var}(\_), \mathit{app}(\_, \_)).}\\
\mathit{neq}(\mathit{var}(\_), \mathit{lam}(\_)). & \ldots &
\end{array}$$

In particular, note that the first clause, because of equivariance, implies that *any* terms $\mathit{var}(\mathsf{a})$ and $\mathit{var}(\mathsf{b})$ with $\mathsf{a} \neq \mathsf{b}$ are α-inequivalent.

Example 2.2.4 (*let*, pairs, and units). We consider some standard extensions to the pure λ-calculus considered so far. One common extension is to permit expressions $\mathit{let}\ x = t\ \mathit{in}\ u$. In the pure λ-calculus, this expression form is definable as $(\lambda x.u)t$; however, in many languages (particularly call-by-value languages like

ML) the two expressions are inequivalent. In any case, we can add syntax and cases to deal with *let* as follows:

$$\begin{array}{lcl}
\mathit{let} & : & (\mathit{exp}, \langle \mathit{id}\rangle \mathit{exp}) \to \mathit{exp}. \\
\mathit{subst}(\mathit{let}(M_1, \langle \mathsf{y}\rangle M_2), T, \mathsf{x})) & = & \mathit{let}(\mathit{subst}(M_1, T, \mathsf{x}), \langle \mathsf{y}\rangle \mathit{subst}(M_2, T, \mathsf{x})). \\
\mathit{typ}(G, \mathit{let}(M_1, \langle \mathsf{y}\rangle M_2), T) & :- & \mathsf{y} \# G, \mathit{typ}(G, M_1, U), \mathit{typ}([(\mathsf{y}, U)|G], M_2, T).
\end{array}$$

Another standard extension is (surjective) pairing. That is, we consider a binary function symbol $\mathit{pair}(t, u)$ and projection operations $\mathit{pi}_1(t)$, $\mathit{pi}_2(t)$ such that $\mathit{pi}_1(\mathit{pair}(t, u)) = t$, $\mathit{pi}_2(\mathit{pair}(t, u)) = u$. A form of pairing is definable in the (untyped) λ-calculus, but often it is preferable to assume that explicit surjective pairing exists. We can extend the αProlog implementation to deal with pairing as follows:

$$\begin{array}{lcl}
\mathit{pair} & : & (\mathit{exp}, \mathit{exp}) \to \mathit{exp}. \\
\mathit{pi}_1, \mathit{pi}_2 & : & \mathit{exp} \to \mathit{exp}. \\
\mathit{times} & : & (\mathit{ty}, \mathit{ty}) \to \mathit{ty}. \\
\mathit{subst}(\mathit{pair}(M_1, M_2), T, \mathsf{x}) & = & \mathit{pair}(\mathit{subst}(M_1, T, \mathsf{x}), \mathit{subst}(M_2, T, \mathsf{x})). \\
\mathit{subst}(\mathit{pi}_1(M_1), T, \mathsf{x}) & = & \mathit{pi}_1(\mathit{subst}(M_1, T, \mathsf{x})). \\
\mathit{typ}(G, \mathit{pair}(M_1, M_2), \mathit{times}(T_1, T_2)) & :- & \mathit{typ}(G, M_1, T_1), \mathit{typ}(G, M_2, T_2). \\
\mathit{typ}(G, \mathit{pi}_1(E), T_1) & :- & \mathit{typ}(G, E, \mathit{times}(T_1, T_2)).
\end{array}$$

The cases for pi_2 are symmetric. Finally, when pairs and pair types are present, it is often useful to also introduce a unit value and unit type, as a "default" base type. This can be accommodated as follows:

$$\begin{array}{lcl}
\star & : & \mathit{exp}. \\
\mathit{seq} & : & (\mathit{exp}, \mathit{exp}) \to \mathit{exp}. \\
\mathit{unit} & : & \mathit{ty}. \\
\mathit{subst}(\star, \_, \_) & = & \star. \\
\mathit{typ}(\_, \star, \mathit{unit}). & & \\
\mathit{typ}(G, \mathit{seq}(E, E'), T) & :- & \mathit{typ}(G, E, \mathit{unit}), \mathit{typ}(G, E', T).
\end{array}$$

Example 2.2.5 (Closure conversion). We give a final meta-programming example for the λ-calculus: *closure conversion* (also known as *lambda lifting* [63]). In the λ-calculus, function bodies may refer to variables introduced outside the body of the function. This phenomenon makes it difficult to perform code transformations; it is much easier to deal with *closed* functions that refer only to local variables (that is, variables bound at the same point as the function is defined).

Closure conversion is a standard translation phase in functional language compilers in which possibly-open function bodies are converted into closed ones. Specifically, a function is replaced with a pair consisting of a closed function and an environment containing expressions for the nonlocal values to which the function refers. Function calls are rewritten so that the environment and closed function are unpacked and the function is called with the environment and ordinary argument as values. Explicit pairing and projection operations are needed in the

$$\begin{array}{lcl}
\textbf{type}\ ctx = [id]. & & \\
\textbf{func}\ cconv(ctx, exp, exp) & = & exp. \\
cconv([\mathsf{x}|G], var(\mathsf{x}), E) & = & pi_1(E). \\
cconv([\mathsf{x}|G], var(\mathsf{y}), E) & = & cconv(G, var(\mathsf{y}), pi_2(E)). \\
cconv(G, app(T_1, T_2), E) & = & let(cconv(G, T_1, E), \langle \mathsf{c} \rangle \\
 & & \quad app(pi_1(var(\mathsf{c})), \\
 & & \qquad pair(cconv(G, T_2, E), pi_2(var(\mathsf{c}))))). \\
cconv(G, lam(\langle \mathsf{x} \rangle T), E) & = & pair(lam(\langle \mathsf{y} \rangle cconv([\mathsf{x}|G], T, var(\mathsf{y}))), E) \\
 & :- & \mathsf{x} \# G, \mathsf{y} \# G.
\end{array}$$

Figure 2.1: Closure conversion in αProlog

$$\begin{array}{lcl}
?\!- & & cconv([], lam(\langle \mathsf{x} \rangle lam(\langle \mathsf{y} \rangle app(var(\mathsf{y}), var(\mathsf{x})))), \star, E). \\
E & = & pair(lam(\langle \mathsf{y}_{17} \rangle pair(lam(\langle \mathsf{y}_{38} \rangle \\
 & & \quad let(pi_1(var(\mathsf{y}_{38})), \langle \mathsf{c}_{51} \rangle \\
 & & \qquad app(pi_1(var(\mathsf{c}_{51})), \\
 & & \qquad\quad pair(pi_1(pi_2(var(\mathsf{y}_{38}))), \\
 & & \qquad\qquad pi_2(var(\mathsf{c}_{51})))))), var(\mathsf{y}_{17}))), \star)
\end{array}$$

Figure 2.2: Closure conversion example

target language, so we consider a λ-calculus extended with *let*, pairing, and projection operations as shown above. We take the source language to contain only λ-abstraction, variables, and application.

We work in an untyped setting (typed closure conversion [87] is substantially more complicated). The following equations define a closure conversion transformation $C[\![\Gamma \vdash t]\!]e$; here, Γ is a list of variables describing the environment of t, and e is a term representing the environment. If Γ has n elements then it is expected that e is an n-tuple.

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

This function can be translated directly into αProlog as shown in Figure 2.1. Figure 2.2 shows a small example: closure-converting the expression $\lambda x.\lambda y.x\ y$. It is not very readable. In more compact notation, this term is expressed as

$$E = (\lambda y_{17}.(\lambda y_{38}.let\ c_{51} = \pi_1(y_{38})\ in\ (\pi_1(c_{51}))\ (\pi_1(\pi_2(y_{38})), \pi_2(c_{51})), y_{17}), \star)$$

which is the correct result of closure-converting $C[\![\cdot \vdash \lambda x.\lambda y.x\ y]\!]$.

```
type cell = (refty, exp).
type mem = [cell].
type config = (mem, exp).

pred update((A, B), [(A, B)], [(A, B)]).
update((a, B), [(a, _)|L], [(a, B)|L]).
update((a, B), [(a', B')|L], [(a', B')|L'])        :-  update((a, B), L, L').

% A call-by-value semantics
pred eval(config, config).
eval((C, ref(A)), (C, ref(A))).
eval((C, new_ref(M)), ([(a, V)|C'], ref(a)))  :-  a # V, a # C',
                                                  eval((C, M), (C', V)).
eval((C, assign(M1, M2)), (C', ref(A)))       :-  eval((C, M1), (C1, ref(A))),
                                                  eval((C1, M2), (C2, V)),
                                                  update((A, V), C2, C').
eval((C, deref(M)), (C, V))                   :-  eval((C, M), (C', ref(A))),
                                                  mem((A, V), C').
```

Figure 2.3: Partial λ_{ref} Implementation

Example 2.2.6. We now consider the problem of modeling a simple λ-calculus with references (see for example [107, Chapter 13]). Following the ML approach, we include three new language syntax cases:

$$e ::= \cdots \mid new(e) \mid !e \mid e := e'$$

represented in αProlog using the term constructors $new\_ref : exp \to exp$, $deref : exp \to exp$, and $assign : (exp, exp) \to exp$.

Figure 2.3 shows part of an αProlog implementation of λ_{ref}. In this example we ignore typechecking issues and focus on the operational semantics of references. The standard operational semantics of references requires considering both the term being evaluated and the contents of memory, often modeled as a finite partial function from some set of *memory cell identifiers* to values. In αProlog, it is natural to use a name type *refty* for these identifiers or references. We use a list of pairs to model the memory itself. In addition we introduce a fourth new expression constructor, $ref : refty \to exp$ that can be used to treat a reference as an expression. Note that such explicit, hardwired references are not allowed in source programs in ML-like languages, but only appear during execution. (In a language like C, references may be explicit memory addresses, but this feature is rarely used except in very low-level systems programming tasks, so this is a reasonable assumption for many C programs also.)

We represent the contents of memory using an association list M containing

$$\begin{array}{lcl}
?\!\!- \ M & = & let(new\_ref(const(17)), \langle \mathsf{x}\rangle \\
 & & \qquad seq(assign(var(\mathsf{x}), const 42), \\
 & & \qquad var(\mathsf{x}))), \\
 & & eval(([], M), (C, M')). \\
M' & = & ref(\mathsf{a}_{68}) \\
C & = & [(\mathsf{a}_{68}, const(42))]
\end{array}$$

$$\begin{array}{lcl}
?\!\!- \ M & = & let(new\_ref(const(17)), \langle \mathsf{x}\rangle let(var(\mathsf{x}), \langle \mathsf{y}\rangle \\
 & & \qquad seq(assign(var(\mathsf{x}), const(42)), \\
 & & \qquad var(\mathsf{y})))), \\
 & & eval(([], M), (C, M')). \\
M' & = & ref(\mathsf{a}_{68}) \\
C & = & [(\mathsf{a}_{68}, const(42))]
\end{array}$$

$$\begin{array}{lcl}
?\!\!- \ M & = & let(new\_ref(const(0)), \langle \mathsf{x}\rangle \\
 & & \qquad let(new\_ref(var(\mathsf{x})), \langle \mathsf{y}\rangle \\
 & & \qquad seq(assign(var(\mathsf{x}), var(\mathsf{y})), \\
 & & \qquad var(\mathsf{x})))), \\
 & & eval(([], M), (C, M')). \\
M' & = & ref(\mathsf{a}_{68}) \\
C & = & [(\mathsf{a}_{143}, ref(\mathsf{a}_{68})), (\mathsf{a}_{68}, ref(\mathsf{a}_{143}))]
\end{array}$$

Figure 2.4: Example queries for λ_{ref}

pairs (r, v) of references r and values v. We define a predicate *update* for updating the value associated to a name in M. Note that *update* requires equivariance to run properly. The predicate *eval* encodes a call-by-value large-step evaluation relation for λ-terms with references. Of particular interest are the cases for *ref*, *new_ref*, *deref*, and *assign*.

The case for *ref* does nothing: a reference is a value. The case for *new_ref* generates a fresh name a for the new memory cell, and assigns it the result of evaluating the given expression. The case for *assign* evaluates its first argument to a name, and updates the memory to the value of its second argument. Finally, *deref* evaluates its argument to a reference and looks up its value.

This is an entirely standard operational semantics for references. The novelty here is the handling of memory references using αProlog names and freshness to enforce the freshness side-condition for names generated by *new_ref*. Note that name-abstraction is never used for reference cells, only freshness.

Figure 2.4 shows three example queries executed in αProlog. The first example allocates a reference with one value and then updates it to take a new value. The second example allocates a reference and creates two aliases to it, updates one of the aliases and evaluates the second. The third example builds a two-node cycle using references.

Table 2.4: The π-calculus

Process terms	$p \quad ::= \quad 0 \mid \tau.p \mid p\vert q \mid p+q \mid x(y).p \mid \bar{x}y.p \mid [x=y]p \mid \nu(x)p$
Actions	$a \quad ::= \quad \tau \mid x(y) \mid \bar{x}y \mid \bar{x}(y)$

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

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

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

There is arguably nothing deep about this example. However, in any ordinary programming language it would have been necessary to either introduce a side-effecting *gensym* function to come up with new names, or clutter the *eval* predicate with added parameters for a fresh name supply. In the first approach, the declarative transparency of the program is destroyed; in the second approach, the cluttered program is harder to read and harder to prove correct.

2.2.2 The π-Calculus

The π-calculus is a calculus of concurrent, mobile processes. Its syntax (following Milner, Parrow, and Walker [85]) is described by the grammar rules shown in Table 2.4. 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 $\bar{x}y.p$ 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 $\nu(y)p$ restricts y to p. Parenthesized names are bound, e.g. y in $x(y).p$ and $\nu(y)p$ are bound in p, 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 Table 2.4, 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, $\bar{x}y$, $x(y)$, $\bar{x}(y)$ 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 $\bar{x}(y)$.

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

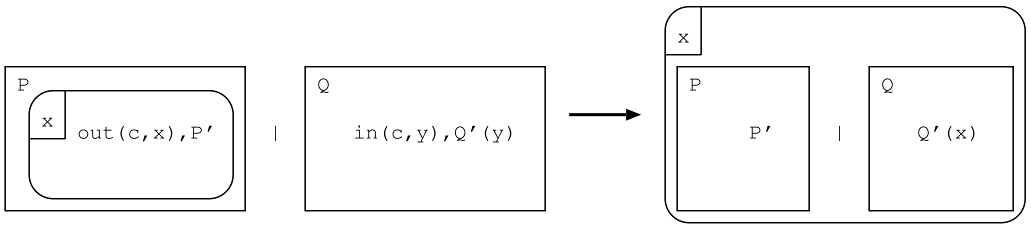

Figure 2.5: Scope extrusion

(Figure 2.5), which occurs when restricted names "escape" their scope because of communication. In $((x)\bar{a}x.p)|(a(z).z(x).0) \xrightarrow{\tau} (x')(p|x'(x).0))$, 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.

In αProlog, processes and actions can be encoded using the following syntax:

$$\begin{array}{llll} chan : \mathbf{name\_type}. & proc : \mathbf{type}. & ina : proc. & tau : proc \to proc. \\ par, sum : (proc, proc) \to proc. & & in : (chan, \langle chan \rangle proc) \to proc. & \\ out, match : (chan, chan, proc) \to proc. & & nu : (\langle chan \rangle proc) \to proc. & \\ act : \mathbf{type}. & tau\_a : act. & in\_a, fout\_a, bout\_a : (chan, chan) \to act. & \end{array}$$

The labeled transition rules can be translated directly into αProlog (see Figure 2.6 and Figure 2.7). The function

$$\mathbf{func}\ ren\_p(proc, chan, chan) \quad = \quad proc.$$

performing capture-avoiding renaming is not shown, but easy to define. The predicate *safe* defined in Figure 2.6 implements the relation $bn(a) \cap fn(q) = \varnothing$ holding between a name a and process q needed in the second rule in Table 2.4.

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

$?\!\!- step(nu(\langle \mathsf{x} \rangle par(nu(\langle \mathsf{y} \rangle out(\mathsf{x}, \mathsf{y}, ina)), in(\mathsf{x}, \langle \mathsf{z} \rangle out(\mathsf{z}, \mathsf{x}, ina)))), A, P).$
$A = tau\_a, P = nu(\langle \mathsf{y}_{58} \rangle nu(\langle \mathsf{z}_{643} \rangle par(ina, out(\mathsf{z}_{643}, \mathsf{y}_{58}, ina))))$
$?\!\!- step(nu(\langle \mathsf{x} \rangle out(\mathsf{x}, \mathsf{y}, ina)), A, P).$
$No.$

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

2.2.3 Additional Examples

The example programs given so far are just a small sample of the test cases I have developed for αProlog. Additional examples that have been developed using αProlog include the operational semantics and type system of Parigot's $\lambda\mu$

$$\begin{array}{lll}
\textbf{pred}\ \mathit{safe}(\mathit{act},\mathit{pr}). & & (\text{* tests } bn(A) \cap fn(P) = \varnothing \text{ *}) \\
\mathit{safe}(\mathit{tau\_a}, P). & & \\
\mathit{safe}(\mathit{fout\_a}(X,Y), P). & & \\
\mathit{safe}(\mathit{bout\_a}(X,\mathsf{y}), P) & :- & \mathsf{y} \mathrel{\#} P. \\
\mathit{safe}(\mathit{in\_a}(X,\mathsf{y}), P) & :- & \mathsf{y} \mathrel{\#} P.
\end{array}$$

Figure 2.6: Safety side-condition

$$\begin{array}{lll}
\textbf{pred}\ \mathit{step}(\mathit{pr},\mathit{act},\mathit{pr}). & & (\text{* encodes } p \xrightarrow{a} p' \text{ *}) \\
\mathit{step}(\mathit{tau}(P), \mathit{tau\_a}, P). & & \\
\mathit{step}(\mathit{par}(P,Q), A, \mathit{par}(P',Q)) & :- & \mathit{step}(P,A,P'), \mathit{safe}(A,Q). \\
\mathit{step}(\mathit{par}(P,Q), \mathit{tau\_a}, \mathit{par}(P',Q'')) & :- & \mathit{step}(P, \mathit{fout\_a}(X,Y), P'), \\
 & & \mathit{step}(Q, \mathit{in\_a}(X,Z), Q'), \\
 & & Q'' = \mathit{ren\_p}(Q', Y, Z). \\
\mathit{step}(\mathit{sum}(P,Q), A, P') & :- & \mathit{step}(P,A,P'). \\
\mathit{step}(\mathit{out}(X,Y,P), \mathit{fout\_a}(X,Y), P). & & \\
\mathit{step}(\mathit{in}(X, \langle \mathsf{z} \rangle P), \mathit{in\_a}(X,\mathsf{w}), P') & :- & \mathsf{w} \mathrel{\#} \langle \mathsf{z} \rangle P, P' = \mathit{ren\_p}(P,\mathsf{w},\mathsf{z}). \\
\mathit{step}(\mathit{match}(X,X,P), A, P') & :- & \mathit{step}(P,A,P'). \\
\mathit{step}(\mathit{par}(P,Q), \mathit{tau\_a}, \mathit{nu}(\langle \mathsf{z} \rangle \mathit{par}(P',Q'))) & :- & \mathit{step}(P, \mathit{bout\_a}(X,\mathsf{z}), P'), \\
 & & \mathit{step}(Q, \mathit{in\_a}(X,\mathsf{z}), Q'). \\
\mathit{step}(\mathit{nu}(\langle \mathsf{y} \rangle P), A, \mathit{nu}(\langle \mathsf{y} \rangle P')) & :- & \mathsf{y} \mathrel{\#} A, \mathit{step}(P,A,P'). \\
\mathit{step}(\mathit{nu}(\langle \mathsf{y} \rangle P), \mathit{bout\_a}(X,\mathsf{w}), P'') & :- & \mathit{step}(P, \mathit{fout\_a}(X,\mathsf{y}), P'), \mathsf{y} \mathrel{\#} X, \\
 & & \mathsf{w} \mathrel{\#} \langle \mathsf{y} \rangle P, P'' = \mathit{ren\_p}(P', \mathsf{w}, \mathsf{y}).
\end{array}$$

Figure 2.7: π-calculus transitions in αProlog

calculus [100], the operational semantics of the ζ-calculus [3], the spi-calculus [2], prenex normalization and Skolemization of first-order logic formulas [21], efficient first-order unification [72], natural deduction systems for first-order and dynamic logics [56] and various type theories, symbolic differentiation [21], constructing automata from regular expressions and removing dead states from automata [57], parsing programming and natural languages to nominal abstract syntax representations using DCGs [101], and programming λ-DRT, an advanced language for computational linguistics [65].

2.3 Notes

Clocksin and Mellish [21] is the standard introductory text for Prolog. The Prolog Standard Reference Manual [31] specifies the behavior of standard Prolog. Pfenning [103] is a collection of work on type systems in logic programming; αProlog uses a ML-style static type system with parametric polymorphism, similar to Mercury's type system [120], but this is only one of many approaches. The type system of αProlog is based on those described by Mycroft and O'Keefe [89] and of Hanus [49]. The unification algorithm used in αProlog is the Nominal Unification algorithm of Urban et al. [126].

The standard reference on the λ-calculus is Barendregt [10]. Closure conversion and lambda lifting are covered in any textbook on compiling functional languages such as [7]. The λ-calculus with references λ_{ref} and the π-calculus are typical examples of programs that are difficult to deal with using higher-order abstract syntax in a logical framework like Twelf, but that can be handled using the *Linear Logical Framework* [15] or *Concurrent Logical Framework* [130].

This chapter expands upon Cheney and Urban [17, 18], where the λ-calculus, $\lambda\mu$-calculus, and π-calculus examples were first given.

Chapter 3

Nominal Sets

A scientific theory should be as simple as possible, but no simpler.

—Albert Einstein

The FM-set-theoretic approach to nominal abstract syntax originally developed by Gabbay and Pitts [42, 43] is powerful, but also has several disadvantages. First, classical set theory is often not a good match for computational applications due to its high theoretical complexity and reliance on nonconstructive reasoning. Second (and somewhat inconsistently with the first part), the Axiom of Choice is not valid in FM-set theory; however, the proof of completeness for first-order logic relies on it. More generally, working in an alternative set theory such as FM is not a step to be taken lightly, and it would be an advantage to work instead in the more familiar universe of ZFC.

Therefore, in this chapter we present an alternative development of nominal abstract syntax, using a class of ordinary mathematical structures which we (following Pitts [108]) call *nominal sets*. A nominal set is a set whose elements admit a sensible *swapping* operation and notion of *support*, such that for every element, a *fresh name* not in the element's support can always be found. Nominal sets are a special case of a well-known algebraic structure: G-sets, or sets acted upon by a group G. We first review the definition of a G-set, then show how nominal sets are obtained as a special case.

The main difference between our account of nominal sets and that of Pitts is that our nominal sets may contain supports that are "infinite but small" whereas Pitts required supports to be finite. This restriction caused problems in the semantics of nominal logic which are solved by our more general development (as shall be shown in Chapter 5.

3.1 Groups and Group Actions

We assume familiarity with the definition of a group and basic concepts from permutation group theory.

Definition 3.1.1 (G-sets). *Given a group $G = (|G|, \mathsf{id}, \circ, ^{-1})$, a G-*set *is a structure $X = (|X|, \cdot_X : G \times |X| \to |X|)$ satisfying $\mathsf{id} \cdot_X x = x$ and $(g \circ h) \cdot_X x = g \cdot_X h \cdot_X x$.*

We say G acts faithfully *on X if whenever $g \cdot_X x = h \cdot_X x$ for every $x \in X$, we have $g = h$.*

Given a G-set X, the orbit *of $x \in X$ is the subset $x^G = \{g \cdot_X x \mid g \in G\}$.*

Given a G-set acting faithfully on X, the support *of $g \in G$ is*

$$support(g) = \{x \in \mathbb{A} \mid g(x) \neq x\}$$

Remark 3.1.2 (Abuse of notation). By an abuse of notation, we often confuse the structure X with its carrier $|X|$, and when no ambiguity ensues we omit subscripts on $\cdot_X$.

Note that the orbits of elements of X partition X into equivalence classes, and the corresponding equivalence relation on X is $x \equiv_G y \iff \exists g \in G. x = g \cdot_X y$.

Remark 3.1.3 (Some G-sets). Before proceeding, we note that G-sets are closed under standard constructions such as products, disjoint union, and power set constructions. For example, if X is a G-set, then $\mathcal{P}(X)$ is a G-set with the natural action inherited by X, such that if $S \in \mathcal{P}(X)$ then

$$g \cdot S = \{g \cdot x \mid x \in S\} .$$

Any ordinary set X encountered in mathematics (e.g. $\mathbb{N}$, $\mathbb{R}$, etc.), equipped with the trivial swapping operation $(a\ a') \cdot_X x = x$ can be viewed as a G-set.

In addition, any group acts on itself by conjugation, that is, G can be viewed as a G-set where

$$g \cdot_G h = g \circ h \circ g^{-1} .$$

We have a specific class of groups G in mind, which we call name-groups.

Definition 3.1.4 (Name-groups). *We assume $\mathbb{A}$ is a given countably infinite G-set, called the set of* names*. Then G is called a name-group if*

1. *G acts faithfully on $\mathbb{A}$,*

2. *G partitions $\mathbb{A}$ into countable orbits $A_1, A_2, \ldots$, also called* name-sets*,*

3. *for each name-set A and $a, b \in A$, there is an element $(a\ b) \in G$ such that*

$$(a\ b) \cdot a = b \qquad (3.1)$$

$$(a\ b) \cdot b = a \qquad (3.2)$$

$$(a\ b) \cdot c = c \qquad (a \neq c \neq b) \qquad (3.3)$$

This element is unique (because G is faithful) and we call it the transposition of a and b.

If, in addition, G satisfies

4. The support of each $g \in G$ is finite.

then G is called a finitary name-group.

In the rest of this chapter, we let $\mathbb{G}$ be a fixed finitary name-group.

Remark 3.1.5. It is not difficult to show that a finitary name-group is isomorphic to the permutation group consisting of all finite permutations on $\mathbb{A}$ respecting the name-sets A. For example, let $A_1, A_2, \ldots$ be a partition of $\mathbb{A}$ into countable sets, and let G be the group generated by all the compatible swappings $(a\ b)$ where $a, b \in A_i$ for some i. Then

$$G = \{g \in FSym(\mathbb{A}) \mid \forall i.g \cdot A_i = A_i\} .$$

It would also be sensible to allow more general name-groups, satisfying only properties 1–3, which would permit infinite permutations. For example, $Sym(A_1) \times Sym(A_2) \times \cdots$, the products of the full transformation groups on the name-sets A, also satisfies properties 1–3. However, this is not necessary for the form of nominal logic considered in this dissertation. This more general class of name-groups would be needed if we considered a form of nominal logic with explicit group elements. Instead, however, we consider only swappings of pairs of compatible names. This simplifies matters considerably.

Note that the name-sets are $\mathbb{G}$-subsets of $\mathbb{A}$ (that is, subsets closed under the restricted action on $\mathbb{A}$). In addition, if $a, a' \in A$, $b, b' \in A'$, and $A \neq A'$ then $(a\ a') \cdot b = b$ since distinct orbits are disjoint.

We say that $a, a' \in \mathbb{A}$ are *compatible* if a and a' are in the same name-set. Let $\mathbb{A}^{[2]}$ be the set of all transpositions of compatible names, that is, $\mathbb{A}^{[2]} = \{(a\ a') \mid A \subset \mathbb{A}, a, a' \in A\}$.

Proposition 3.1.6 ($\mathbb{G}$-set laws). *Suppose $X = (|X|, \cdot_X)$ is a $\mathbb{G}$-set. Then*

$$\begin{aligned} (a\ a) \cdot_X x &= x && (3.4)\\ (a\ a') \cdot_X (a\ a') \cdot_X x &= x && (3.5)\\ (a\ a') \cdot_X (b\ b') \cdot_X x &= ((a\ a') \cdot b\ (a\ a') \cdot b') \cdot_X (a\ a') \cdot_X x && (3.6) \end{aligned}$$

Conversely, if $|X|$ is a set equipped with a swapping function $(a\ b), x \to (a\ b) \cdot x : \mathbb{A}^{[2]} \times X \to X$ satisfying the above laws, then the swapping action can be extended to a full action $\cdot_X$ of $\mathbb{G}$ on $|X|$ making $X = (|X|, \cdot_X)$ a $\mathbb{G}$-set.

Proof. The first part is an immediate consequence of the definition of transpositions. The second part relies on the fact that since $\mathbb{G}$ is finitary, it is generated by swappings of compatible names, i.e. by the elements of $\mathbb{A}^{[2]}$. □

3.2 Equivariance, Support and Freshness

3.2.1 Equivariance

As might be expected, there is a natural criterion for a function mapping one G-set to another to be structure-preserving:

Definition 3.2.1 (Equivariance). *Given G-sets X, Y, a function $f : X \to Y$ is* equivariant *from X to Y provided for any $x \in X$ and $g \in G$,*

$$g \cdot_Y f(x) = f(g \cdot_X x) \; . \tag{3.7}$$

A relation $R \subseteq |X|$ on a G-set X is an equivariant relation *on X provided for any $x \in X$, $g \in G$,*

$$R(x) \iff R(g \cdot_X x) \; . \tag{3.8}$$

A value $x \in X$ is equivariant *provided for any $g \in G$,*

$$g \cdot_X x = x \; . \tag{3.9}$$

3.2.2 Support Ideals

Equivariance means invariance under arbitrary swappings. This is a special case of a more general phenomenon: invariance under swapping all but a certain set of names (often called a *support*). Intuitively, a support of an element x of a $\mathbb{G}$-set X is a subset of $\mathbb{A}$ that contains all the names "used" or "mentioned" in x. In group theory, the support of a permutation is the set of elements moved by the permutation. In a $\mathbb{G}$-set, the (intuitive) notion of a support of x is a set S such that swapping any two names *not* in S fixes x, or more generally,

for every finite permutation π fixing every element of S, π fixes x.

Pitts' original semantics for nominal logic prescribed a *finite support property*: all elements of nominal sets were required to have finite supports. Unfortunately, as we shall see (Chapter 5), nominal logic is incomplete with respect to this semantics because (intuitively) the statement "there exists a value with infinite support" can be expressed in nominal logic but the statement "every value has finite support" cannot. Since statements about finiteness usually cannot be expressed in first-order logic, there is little reason to believe that this problem can be repaired while remaining within an essentially first-order framework.

However, without some restriction on which sets can be supports, minimum supports may not even exist. For example, using the naïve definition above, both $\{a\}$ and $\mathbb{A} - \{a\}$ can be considered minimal supports for $a \in \mathbb{A}$, since $(b\ b') \cdot a = a$ precisely when $b = b' = a$ or $b \neq a \neq b'$. Clearly, however, the preferred support of a is $\{a\}$. When a finite support exists, it is clearly preferable to an infinite support; on the other hand, for a value with only infinite supports, it may be difficult to choose. For example, if $x = \{a_1, a_3, \ldots\}$ consists of the odd-numbered names, then

$\{x\}$ supports x but so does $\mathbb{A} - x$ (and both x and $\mathbb{A} - x$ also support $\mathbb{A} - x$). In this case the choice of "real" support seems arbitrary.

Therefore, we adopt the position that some *a priori* choice concerning which sets are acceptable as supports must be made. The intuition is that supports should be "small" in some sense (not necessarily in terms of cardinality). Clearly $\varnothing$ and singletons $\{a\}$ (or more generally, finite sets of names) are small. Also, it seems reasonable to expect that subsets and finite unions of small sets are small. Since we want to ensure that fresh names of any sort can always be chosen, A should *not* be a support for any name-set A. Finally, if S is a support then for any $g \in \mathbb{G}$, $g \cdot S$ should also be a support. (For finitary name-groups, this property is derivable from the others.)

These properties suggest that the set of all supports forms a special kind of *ideal*, defined as follows:

Definition 3.2.2. *A* support ideal *$\mathcal{I}$ is a collection of subsets of $\mathbb{A}$ satisfying*

1. *If $S \subseteq T \in \mathcal{I}$ then $S \in \mathcal{I}$.*
2. *If $S, T \in \mathcal{I}$ then $S \cup T \in \mathcal{I}$.*
3. *If $a \in \mathbb{A}$ then $\{a\} \in \mathcal{I}$.*
4. *$A \notin \mathcal{I}$ for any name-set A.*

We refer to the sets in $\mathcal{I}$ as small *(with respect to $\mathcal{I}$) and the subsets of $\mathbb{A}$ not in $\mathcal{I}$ as* large *(with respect to $\mathcal{I}$).*

Proposition 3.2.3. *If $\mathcal{I}$ is a support ideal and a, b are compatible and $S \in \mathcal{I}$ then $(a\ b) \cdot S \in \mathcal{I}$.*

Proof. There are two cases: if $a, b \in S$ or $a, b \notin S$ then $(a\ b) \cdot S = S \in \mathcal{I}$. Otherwise without loss of generality assume $a \in S, b \notin S$. Then $(a\ b) \cdot S = (S - \{a\}) \cup \{b\}$. By (1), $S - \{a\} \subseteq S \in \mathcal{I}$ so $S - \{a\} \in \mathcal{I}$. By (3), $\{b\} \in \mathcal{I}$, and by (2), $(S - \{a\}) \cup \{b\} \in \mathcal{I}$. □

Proposition 3.2.4. *The set $\mathcal{P}_{<\omega}(\mathbb{A}) = \{S \subseteq \mathbb{A} \mid |S| < \omega\}$ of all finite subsets of $\mathbb{A}$ forms a support ideal.*

Proof. $\mathcal{P}_{<\omega}(\mathbb{A})$ contains all singletons, is obviously closed under subset and union, and A is not finite for any A. □

Is $\mathcal{P}_{<\omega}(\mathbb{A})$ the *only* support ideal over a countable $\mathbb{A}$? The answer is no. One counterexample is the set of *sparse* (that is, nowhere dense) subsets of $\mathbb{Q}$ with the usual open interval topology.

Proposition 3.2.5. *There exists a support ideal over a countable $\mathbb{A}$ containing a countable element.*

Proof. Consider the standard open-interval topology of $\mathbb{Q}$. We identify $\mathbb{A}$ with $\mathbb{Q}$ in some arbitrary way such that each name-set A is dense, and say that $S \subseteq \mathbb{A}$ is *sparse* if it has no accumulation points. Let $\mathcal{I}$ be the set of all sparse subsets of $\mathbb{A}$.

- If $S \subseteq T \in \mathcal{I}$ then any accumulation point of S would also be an accumulation point of T, so $S \in \mathcal{I}$.
- If $S, T \in I$ then any accumulation point of $S \cup T$ would also have to be an accumulation point of S or of T, so $S \cup T \in \mathcal{I}$.
- If $a \in \mathbb{A}$, then $\{a\} \in \mathcal{I}$ is obviously sparse.
- Finally, as observed before, no name-set A is sparse.

This shows that $\mathcal{I}$ is a support ideal. Clearly, there are infinite sparse subsets of $\mathbb{A}$, so $\mathcal{I}$ contains infinite supports. In fact, it contains infinite supports within each A. □

3.2.3 Support and Nominal Sets

We now can give a general definition of support. In this section, let $\mathcal{I}$ be a fixed support ideal.

Definition 3.2.6 (Support). *Let X be a $\mathbb{G}$-set, $x \in X$, $w \subseteq \mathbb{A}$. We say that w* supports *x ($w \lhd x$) if*

$$\forall a, a' \in \mathbb{A}^{[2]} - w^2.(a\ a') \cdot_X x = x\ . \tag{3.10}$$

Lemma 3.2.7. *Suppose $w \lhd x$. If $a, b \notin w$ then $(a\ b) \cdot x = x$. Conversely, if $(a\ b) \cdot x \neq x$ then $a \in w$ or $b \in w$.*

Proof. The first part is immediate from the definition of $\lhd$. The second part is the contrapositive. □

Lemma 3.2.8. *Suppose X is a $\mathbb{G}$-set and $w, w' \lhd x \in X$ with $w, w' \in \mathcal{I}$. Then $w \cap w' \in \mathcal{I}$ supports x as well.*

Proof. Clearly $w \cap w' \in \mathcal{I}$. Let $(a\ a') \in \mathbb{A}^{[2]} - (w \cap w')^2$ be given. If a, a' both lie in the complement of w (symmetrically, w'), then clearly $(a\ a') \cdot x = x$. Otherwise, without loss of generality assume $a \notin w$ and $a' \notin w'$. Since $w, w' \in \mathcal{I}$, we must also have $w \cup w' \in \mathcal{I}$, so we may choose $b \in A - w \cup w'$. Then $b \in A - w$ and $b \in A - w'$. Since $(a\ a') = (a\ b)(a'\ b)(a\ b)$, we have

$$(a\ a') \cdot x = (a\ b)(a'\ b)(a\ b) \cdot x = x$$

because $(a\ b)$ and $(a'\ b)$ both fix x. □

Proposition 3.2.9. *Let X be a $\mathbb{G}$-set and $x \in X$ be given, and assume x has a support in $\mathcal{I}$. Then x has a $\subseteq$-least support in $\mathcal{I}$.*

Proof. Let $S = \bigcap\{w \in \mathcal{I} \mid w \lhd x\}$. If x has a least support in $\mathcal{I}$, then it must be S. We wish to show that $S \lhd x$.

Suppose $a, a' \in \mathbb{A} - S$. Then there exist $w, w' \lhd x$ such that $a \notin w$ and $a' \notin w'$. Without loss of generality, we can assume that w and w' are small. By Lemma 3.2.8, $w \cap w' \lhd x$ and $a, a' \notin w \cap w'$; hence $(a\ a') \cdot x = x$. This completes the proof. □

Definition 3.2.10 (Nominal sets). *A* nominal set *is a $\mathbb{G}$-set all of whose elements are supported in $\mathcal{I}$. Given a nominal set, we define the* support of $x \in X$ *($supp_X(x)$) as*

$$supp_X(x) = \bigcap\{w \in \mathcal{I} \mid w \lhd x\}\ . \tag{3.11}$$

Proposition 3.2.11 (Equivalent views of *supp***).** *The following are equivalent:*

1. *$a \in supp(x)$.*
2. *For any $b \notin supp(x)$, $(a\ b) \cdot x \neq x$*
3. *For some $b \notin supp(x)$, $(a\ b) \cdot x \neq x$*

Proof. We prove that $1 \to 2$, $2 \to 3$, and $3 \to 1$.

- $(1 \to 2)$: Assume $a \in supp(x)$ and $b \notin supp(x)$. Assume for contradiction that $(a\ b) \cdot x = x$. Then we will show that $supp(x) - \{a\}$ supports x. Let $a', b' \notin supp(x) - \{a\}$ be given. If neither or both a', b' is a, there is nothing to prove since $(a'\ b') \cdot x = x$. Otherwise, assume $a' = a$. Then $(a\ b') \cdot x = (b\ b') \cdot (a\ b) \cdot (b\ b') \cdot x$, and since both $b, b' \notin supp(x)$ and by assumption $(a\ b) \cdot x = x$, we have $(b\ b') \cdot (a\ b) \cdot (b\ b') \cdot x = x$. This shows that $supp(x) - \{a\}$ supports x, contradicting the minimality of $supp(x)$.

- $(2 \to 3)$: Assume that for any $b \notin supp(x)$, $(a\ b) \cdot x \neq x$. There must be a name-set A such that $a, b \in A$. Since supports are always elements of $\mathcal{I}$ and $A \notin \mathcal{I}$, there must be an element b of A not in $supp(x)$, and by assumption any such b satisfies $(a\ b) \cdot x \neq x$.

- $(3 \to 1)$: Assume that $b \notin supp(x)$ and $(a\ b) \cdot x \neq x$. By the contrapositive of the definition of support, if $(a\ b) \cdot x \neq x$ then $a \in supp(x)$ or $b \in supp(x)$, but clearly $b \notin supp(x)$ so we can conclude $a \in supp(x)$.

□

Remark 3.2.12 (Some nominal sets). Clearly, any name-set A is a nominal set since $\{a\} \in \mathcal{I}$ supports a. Also, $w \lhd w$ for any $w \in \mathcal{I}$ so $\mathcal{I}$ is a nominal set. Finally, for the natural action of $\mathbb{G}$ on itself by conjugation, we have $supp_{\mathbb{G}}(g) = support(g)$ where the latter is as defined in Definition 3.1.4. Since every element of $\mathbb{G}$ has finite support, $\mathbb{G}$ is a nominal set. We now prove these facts.

Proposition 3.2.13. *1. A name-set A is a nominal set and $\mathit{supp}_A(a) = \{a\}$ if $a \in A$.*

2. $\mathcal{I}$ is a nominal set and $\mathit{supp}_{\mathcal{I}}(S) = S$.

3. $\mathbb{G}$ is a nominal set and $\mathit{supp}_{\mathbb{G}}(g) = \mathit{support}(g) = \{a \in \mathbb{A} \mid g \cdot_{\mathbb{A}} a \neq a\}$.

4. Let X be a nominal set. Then $\mathit{supp}_X(x) = \varnothing$ if and only if x is equivariant.

Proof. 1. Obviously $\{a\} \lhd a$ since if $b, b' \notin \{a\}$ then $(b\ b') \cdot a = a$. On the other hand, $\varnothing$ does not support a, so $\mathit{supp}(a) = \{a\}$. So A is a nominal set and $\mathit{supp}_A(a) = \{a\}$.

2. First, note that it is obvious that $S \lhd S$. We only need to show that S is the least support of S in $\mathcal{I}$, i.e. that $S \subseteq T$ if $T \in \mathcal{I}$ supports S. We show the contrapositive. If $S \not\subseteq T$ then choose $a \in S - T$, which must be nonempty. Since $S, T \in \mathcal{I}$, there must be an element $b \in \mathbb{A} - S \cup T$ compatible with a. Observe that $(a\ b) \cdot S = (S - \{a\}) \cup \{b\}$ since $a \in S$ and $b \notin S$; however, $a \notin T$ and $b \notin T$, so T cannot be a support of S.

3. First, observe that $\mathit{support}(g) \lhd g$, for if $a, b \in \mathbb{A}^{[2]} - \mathit{supp}(g)^2$, g fixes both a and b, so g commutes with $(a\ b)$. Also, $\mathit{support}(g)$ is finite so must be in $\mathcal{I}$, consequently $\mathbb{G}$ is a nominal set. We have

$$(a\ b) \cdot g = (a\ b) \circ g \circ (a\ b) = (a\ b) \circ (a\ b) \circ g = g\ .$$

Moreover, if $a \in \mathit{support}(g)$ and $b \notin \mathit{support}(g)$ then let $c = g \cdot a$. Note that $c \neq a$ (since $a \in \mathit{support}(g)$) and $c \neq b$ (since $c \in \mathit{support}(g)$ and $b \notin \mathit{support}(g)$). We have

$$((a\ b) \cdot g) \cdot_{\mathbb{A}} b = (a\ b) \cdot (g \cdot_{\mathbb{A}} a) = (a\ b) \cdot c = c$$

hence, $(a\ b) \cdot g \neq g$ since g fixes b. This shows that if $a \in \mathit{support}(g)$ then $a \in \mathit{supp}_{\mathbb{G}}(g)$, by Proposition 3.2.11. Thus, $\mathit{support}(g) \subseteq \mathit{supp}_{\mathbb{G}}(g)$. Since $\mathit{supp}_{\mathbb{G}}(g)$ is the least support, $\mathit{supp}_{\mathbb{G}}(g) = \mathit{support}(g)$.

4. If x is equivariant then for any compatible a, b we have $(a\ b) \cdot x = x$, so $\varnothing \lhd x$. Conversely, if $\mathit{supp}(x) = \varnothing$ then $(a\ b) \cdot x = x$ for every compatible $a, b \in \mathbb{A}$.

□

Lemma 3.2.14. *Let X be a nominal set and A a name-set.*

1. The relation $\lhd \subseteq \mathcal{I} \times X$ is equivariant.

2. The function $\mathit{supp}_X : X \to \mathcal{I}$ is equivariant.

3. The relation $=: X \times X$ is equivariant.

4. *If R is an equivariant relation, then so are its reflexive, symmetric, and transitive closures (and any combination thereof).*

Proof. 1. Suppose $w \lhd x$. Let b, b' be given and set $x' = (b\ b') \cdot x$ and $w' = (b\ b') \cdot w$. Then $\forall a, a' \in \mathbb{A} - w.(a\ a') \cdot x = x$. Let $a, a' \notin \mathbb{A} - w'$ be given. Then $(b\ b') \cdot a, (b\ b') \cdot a' \notin \mathbb{A} - w$, so $((b\ b') \cdot a\ (b\ b') \cdot a') \cdot x = x$. Moreover,

$$\begin{aligned}(a\ a') \cdot x' &= (a\ a') \cdot (b\ b') \cdot x \\ &= (b\ b') \cdot ((b\ b') \cdot a\ (b\ b') \cdot a') \cdot x \\ &= (b\ b') \cdot x \\ &= x' .\end{aligned}$$

Since a, a' were arbitrary, this shows that $\forall a, a' \in \mathbb{A} - w'.(a\ a') \cdot x' = x'$, so $w' \lhd x'$ as desired.

2. For $supp_X((a\ b) \cdot x) = (a\ b) \cdot supp_X(x)$, assume $a' \in supp_X((a\ b) \cdot x)$. We wish to show that $a' \in (a\ b) \cdot supp_X(x)$, or equivalently, that $(a\ b) \cdot a' \in supp_X(x)$. It suffices to show that if $w \lhd x$ then $(a\ b) \cdot a' \in w$. Suppose $w \lhd x$. Clearly, then, $(a\ b) \cdot w \lhd (a\ b) \cdot x$ by part 1. So $a' \in (a\ b) \cdot w$ since $(a\ b) \cdot w \lhd (a\ b) \cdot x$ and $a' \in supp_X((a\ b) \cdot x)$. But this means that $(a\ b) \cdot a' \in w$, as desired. The reverse inclusion is similar.

3. Let a, b and $x = y \in X$ be given. Then $(a\ b) \cdot x = (a\ b) \cdot y$ also.

4. Let R be equivariant. If R_r is its reflexive closure, then xR_ry if $x = y$ or xRy. In either case, we can derive $(a\ b) \cdot xR_r(a\ b) \cdot y$. For R_s the symmetric closure, xR_sy if xRy or yRx, and again in either case we can derive $(a\ b) \cdot xR_s(a\ b) \cdot y$. Finally, for R_t the transitive closure, xR_tz if $x = y_1R \cdots Ry_{n+1} = z$. By induction on n, the number of steps from x to z, we can show that $(a\ b) \cdot x = (a\ b) \cdot y_1R \cdots R(a\ b) \cdot y_{n+1} = (a\ b) \cdot z$ and so $(a\ b) \cdot xR_t(a\ b) \cdot z$

□

3.2.4 Freshness

We are now in a position to define the freshness relation $\#: A \times X$ for any name-set A and nominal set X.

Definition 3.2.15. *A name $a \in \mathbb{A}$ is* fresh *for $x \in X$ (written $a \#_X x$) provided $a \notin supp_X(x)$.*

We write $a \# \vec{x}$ to indicate that $a \# x_1, \ldots, a \# x_n$ all hold.

Proposition 3.2.16 (Properties of freshness). *Let $a, b \in \mathbb{A}$ and nominal set X be given.*

1. *$\#: A \times X$ is an equivariant relation.*

2. *If $a \# x$ and $b \# x$ then $(a\ b) \cdot x = x$.*

3. *$a \# b$ if and only if $a \neq b$.*

4. *$a \# x$ for any equivariant value $x \in X$.*

Proof. Let a, X be as specified.

1. Let $a \in A, x \in X$ with $a \# x$ be given. Suppose b, b' are compatible. Then $a \notin \mathit{supp}(x)$, so $(b\ b') \cdot a \notin (a\ b) \cdot \mathit{supp}(x)$. Note that since *supp* is equivariant, $(b\ b') \cdot \mathit{supp}(x) = \mathit{supp}((a\ b) \cdot x)$. Consequently, $(b\ b') \cdot a \# (a\ b) \cdot x$.

2. If $a \# x$ and $b \# x$ then $a, b \notin \mathit{supp}(x)$ so $(a\ b) \cdot x = x$ since $\mathit{supp}(x) \lhd x$.

3. Observe that $a \notin \mathit{supp}(b) = \{b\}$ if and only if $a \neq b$.

4. If x is equivariant then $\mathit{supp}(x) = \varnothing$ so any $a \notin \mathit{supp}(x)$.

□

3.2.5 Reasoning with Fresh Names

Gabbay and Pitts [42, 43, 108] observed that a novel quantifier, N, may be defined in FM-set theory as

$$\mathsf{N}a.\phi \iff \{a \mid \phi(a)\} \text{ is cofinite}$$

that is, ϕ holds of a *new* a just in case it holds for "almost all" a, or all a except those in a finite set. Moreover, N is self-dual, as can be seen by the equivalences

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

This definition of N is not appropriate for ideal-supported nominal sets because some elements might have infinite (but small) support. Instead, we say that $\phi(a)$ holds for fresh a (or, symbolically, $\mathsf{N}a.\phi(a)$) provided that the set $\{a \mid \phi(a)\}$ is *large* (with respect to $\mathcal{I}$).

In subsequent chapters, we will formalize the properties of the N-quantifier within Nominal Logic. While doing so, however, we shall also need to reason about the fresh names and variables occurring in the syntax of NL.

Remark 3.2.17 (Conventions for reasoning with names and N). In an informal mathematical proof, an argument of the form

> Let a be fresh. [Argument establishing $P(a)$]. Since a was chosen fresh, $P(a)$ holds for any fresh name.

may be used to establish that $\mathsf{N}a.P(a)$ holds. Conversely, if $\mathsf{N}a.P(a)$ holds, then the argument

> Since $\mathsf{N}a.P(a)$ holds, $P(a)$ must hold for some fresh a. [Argument establishing Q.]

In either case, within the arguments enclosed in brackets, a may be assumed to be fresh for any element of a nominal set that was already present in the argument before a was introduced, without explicit mention of this fact. For example:

> Let x be given. Let a be fresh. Then $a \# x$. Since a was fresh, $\text{И}a.a \# x$ holds. Since x was arbitrary, $\forall x.\text{И}a.a \# x$ holds.

These informal proof principles will be given a more formal development in nominal logic.

3.3 Constructions on Nominal Sets

In this section we present some basic nominal sets and constructions for building new nominal sets from existing ones.

3.3.1 Standard Constructions

Definition 3.3.1 (Initial, terminal). *The initial nominal set is* $\mathbf{0}$*, carried by the empty set and the empty swapping operation.*

The terminal nominal set is $\mathbf{1}$*, carried by a singleton set* $\{\star\}$ *with the trivial swapping operation.*

Proposition 3.3.2. *The initial and terminal structures are nominal sets. Moreover,* $supp_{\mathbf{0}}(y) = supp_{\mathbf{1}}(z) = \varnothing$*, and* $x \#_{\mathbf{0}} y$*,* $x \#_{\mathbf{1}} y$ *for any* $x \in \mathbb{A}$*,* $y \in \mathbf{0}$*, and* $z \in \mathbf{1}$*.*

Definition 3.3.3 (Product). *Given nominal sets* X, Y*, the product of* X *and* Y *is* $X \times Y$*, carried by* $|X| \times |Y|$ *and with swapping operation defined by*

$$(a\ b) \cdot_{X\times Y} (x, y) = ((a\ b) \cdot_X x, (a\ b) \cdot_Y y)$$

Proposition 3.3.4. *The product of two nominal sets is a nominal set. Moreover,* $supp_{X\times Y}(x, y) = supp_X(x) \cup supp_Y(y)$*, and* $a \#_{X\times Y} (x, y)$ *if and only if* $a \#_X x$ *and* $a \#_Y y$*.*

Proof. The nominal set axioms are easy to verify.

For the second part, we first show that $supp(x, y) \subseteq supp(x) \cup supp(y)$. If $a, b \notin supp(x) \cup supp(y)$ then $(a\ b) \cdot (x, y) = ((a\ b) \cdot x, (a\ b) \cdot y) = (x, y)$. So $supp(x) \cup supp(y) \lhd (x, y)$; consequently $supp(x, y) \subseteq supp(x) \cup supp(y)$.

We now show that $supp_{X\times Y}(x, y) \supseteq supp_X(x) \cup supp_Y(y)$. If $a \in supp(x) \cup supp(y)$, suppose $a \in supp(x)$ (the case for y is symmetric). Then for any $b \notin supp(x) \cup supp(y)$, $(a\ b) \cdot x \neq x$, and some such b exists, so for some $b \notin supp(x) \cup supp(y)$, $(a\ b) \cdot (x, y) \neq (x, y)$. Hence $a \in supp(x, y)$.

The third part is an immediate consequence of the second. □

Definition 3.3.5 (Sum (or disjoint union)). *Given nominal sets X, Y, the sum of X and Y is $X+Y$, carried by $|X| \uplus |Y| = \{0\} \times X \cup \{1\} \times Y$ and with swapping operation defined by*

$$(a\ b) \cdot_{X+Y} (0, x) = (0, (a\ b) \cdot_X x) \qquad (a\ b) \cdot_{X+Y} (1, y) = (0, (a\ b) \cdot_Y y)$$

Proposition 3.3.6. *The sum of two nominal sets is a nominal set. Moreover, $supp_{X+Y}(0, x) = supp_X(x)$, $supp_{X+Y}(1, y) = supp_Y(y)$, and $a \# (i, z) \iff a \# z$ for $i \in \{0, 1\}$.*

The proof is straightforward and omitted.

Some of the standard constructions on $\mathbb{G}$-sets, such as exponentiation (function space construction) and power set, do not preserve the $\mathcal{I}$-support property. For example, for a name-set A, the full set-theoretic powerset may contain sets that are not $\mathcal{I}$-supported. For example, any name-set A contains infinite infinite subsets S, T such that $S \cup T = A$ and $S \cap T = \varnothing$; it cannot be the case that both S and T are in $\mathcal{I}$. However, we can recover adequate versions of these constructions by filtering out the non-$\mathcal{I}$-supported elements.

Definition 3.3.7 (Restriction). *Given a $\mathbb{G}$-set X, and support ideal $\mathcal{I}$, the restriction $X|_\mathcal{I}$ of X to $\mathcal{I}$ is given by restricting the carrier $|X|$ to*

$$|X|_\mathcal{I}| = \{x \in |X| \mid \exists w \in \mathcal{I}. w \lhd x\}$$

Proposition 3.3.8. *The restriction of a $\mathbb{G}$-set to $\mathcal{I}$ is a nominal set.*

Definition 3.3.9 (Exponential (or function space)). *Given nominal sets X, Y, the* exponential *of X and Y is the set of all $\mathcal{I}$-supported functions from X to Y, written Y^X. This set is carried by the restricted $\mathbb{G}$-exponential $(X \to Y)_\mathcal{I}$, with swapping operation defined by*

$$(a\ b) \cdot_{Y^X} f = x \mapsto (a\ b) \cdot_Y (f((a\ b) \cdot_X x))$$

Proposition 3.3.10. *The exponential of two nominal sets is a nominal set. Moreover, $supp_Y(f(x)) \subseteq supp_{Y^X}(f) \cup supp_X(x)$, and $a \#_Y f(x)$ if $a \#_{Y^X} f$ and $a \#_X x$.*

Proof. The swapping axioms are easy to verify.

We will show that every support of both f and x is a support of $f(x)$; this implies $supp(f(x)) \subseteq supp(f) \cup supp(x)$. Let $w = supp(x) \cup supp(f)$. Then

$$(a\ b) \cdot f(x) = ((a\ b) \cdot f)((a\ b) \cdot x) = f(x)$$

since $(a\ b)$ must fix both f and x. So $w \lhd f(x)$, consequently $supp(f(x)) \subseteq w = supp(f) \cup supp(x)$.

If $f \in Y^X$ then $supp(f(x)) \subseteq supp(f) \cup supp(x)$ so $a \# f, x$ implies $a \# f(x)$. $\square$

Definition 3.3.11 (Power set). *Given a nominal set X, the nominal power set of X, written $\mathcal{P}(X)$, is carried by the restricted power $\mathbb{G}$-set $\mathcal{P}(|X|)|_{\mathcal{I}}$ and the swapping operation on $\mathcal{P}(X)$ is given by*

$$(a\ b) \cdot_{\mathcal{P}(X)} S = \{(a\ b) \cdot_X x \mid x \in S\}$$

Proposition 3.3.12. *The power set of a nominal set is a a nominal set. Moreover, $\in : X \times \mathcal{P}(X)$ and $\subseteq : \mathcal{P}(X) \times \mathcal{P}(X)$ are equivariant relations, and $\bigcup, \bigcap : \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X)$ are equivariant functions.*

Proof. The axioms of swapping are easy to verify, as is the fact that $\mathcal{P}(X)$ is $\mathcal{I}$-supported.

Let a, b and x, y be given with $x \in y$. Then $(a\,b)\cdot x \in (a\,b)\cdot y = \{(a\,b)\cdot z \mid z \in y\}$. Similarly, since $x \subseteq y$ precisely if whenever $z \in x$, we have $z \in y$, it is obvious that $\subseteq$ is equivariant.

Let $\mathcal{C}$ be a collection of subsets of X. We wish to show that $(a\ b) \cdot \bigcup \mathcal{C} = \bigcup (a\ b) \cdot \mathcal{C}$. But

$$\begin{aligned}
(a\ b) \cdot \bigcup \mathcal{C} &= (a\ b) \cdot \{x \mid x \in C, C \in \mathcal{C}\} \\
&= \{(a\ b) \cdot x \mid x \in C, C \in \mathcal{C}\} \\
&= \{(a\ b) \cdot x \mid (a\ b) \cdot x \in (a\ b) \cdot C, (a\ b) \cdot C \in (a\ b) \cdot \mathcal{C}\} \\
&= \{y \mid y \in D, D \in (a\ b) \cdot \mathcal{C}\} \\
&= \bigcup (a\ b) \cdot \mathcal{C}
\end{aligned}$$

The proof for $\bigcap$ is similar. □

Proposition 3.3.13. *If X is a nominal set and Y an equivariant subset of X then Y is a nominal set.*

Proof. The axioms of swapping for Y are special cases of those for X. So, all that is needed is to verify that Y is closed under swapping, i.e., that $x \in Y \iff (a\ b) \cdot x \in Y$ for each x. But if $x \in Y$ then $(a\ b) \cdot x \in (a\ b) \cdot Y = Y$ and vice versa, since Y is equivariant. □

Definition 3.3.14. *Let X, Y be nominal sets with $|X| \subseteq |Y|$. We say $X \subseteq_{\mathbf{Nom}} Y$, or X is a nominal subset of Y, if*

$$(a\ b) \cdot_X x = (a\ b) \cdot_Y x$$

for each $x \in X$.

A chain of nominal sets is an increasing sequence $X_1 \subseteq_{\mathbf{Nom}} X_2 \subseteq_{\mathbf{Nom}} \cdots$.

Proposition 3.3.15. *Suppose $X_1 \subseteq_{\mathbf{Nom}} X_2 \subseteq_{\mathbf{Nom}} \cdots$ is chain of nominal sets. Then $\bigcup_i X_i$ is a nominal set with $\cdot_X = \bigcup_i \cdot_{X_i}$.*

Proof. It is straightforward to verify that $\bigcup_i X_i$ is a $\mathbb{G}$-set with the given swapping operation. To see that every element of $\bigcup_i X_i$ is $\mathcal{I}$-supported, let x be such an element. Then there must exist i such that $x \in X_i$. Since X_i is $\mathcal{I}$-supported, x has a support in $\mathcal{I}$. This completes the proof. $\square$

Definition 3.3.16 (Quotient). *Suppose X is a nominal set and $\equiv\colon X \times X$ is an equivariant relation on X. Then the quotient $X/\equiv$ is the structure consisting of $|X|/\equiv$ and with swapping defined as*

$$(a\ b) \cdot S = \{(a\ b) \cdot x \mid x \in S\}$$

for equivalence classes $S \in X/\equiv$.

Proposition 3.3.17. *The quotient of a $\mathbb{G}$-set or nominal set by an equivariant relation is a $\mathbb{G}$-set or nominal set respectively.*

Proof. For the quotient construction, it suffices to show that for an equivalence class $[x] \in X/\equiv$,

$$(a\ b) \cdot [x] = [(a\ b) \cdot x]\ .$$

To show this, observe

$$\begin{aligned}(a\ b) \cdot [x] &= \{(a\ b) \cdot y \mid y \equiv x\} \\ &= \{(a\ b) \cdot y \mid (a\ b) \cdot y \equiv (a\ b) \cdot x\} \\ &= \{z \mid z \equiv (a\ b) \cdot x\}\end{aligned}$$

where the second equality follows from the equivariance of $\equiv$. Then the swapping laws and the $\mathcal{I}$-support property follow from the corresponding properties of X. For example, any support of x is a support of $[x]$ since if $a, b \notin w \lhd x$ then

$$(a\ b) \cdot [x] = [(a\ b) \cdot x] = [x]$$

This completes the proof. $\square$

3.3.2 Abstractions

With the concept of support and the above constructions in hand, we can define the *abstraction* construction on nominal sets. Intuitively, an abstraction $\langle a \rangle x$ is an element x of a nominal set X with a locally-scoped name a: thus, a can be renamed in $\langle a \rangle x$ uniformly without changing the meaning of $\langle a \rangle x$ as long as there is no collision with the other names of x. That is, abstraction terms are considered equal up to α-equivalent renaming of abstracted name-constants.

Definition 3.3.18. *Given a name-set A and an arbitrary nominal set X, define the relation $\equiv_\alpha\colon (A \times X) \times (A \times X)$ to be the reflexive, symmetric, transitive closure of the relation $\sim_\alpha$:*

$$(a, x) \sim_\alpha (b, y) \textit{ if and only if } x = (a\ b) \cdot_X y \textit{ and } a \# y\ .$$

Proposition 3.3.19. *$\equiv_\alpha$ is an equivariant equivalence relation.*

Proof. Note that $\sim_\alpha$ is defined in terms of equality and freshness, both of which are equivariant relations, and so is equivariant. By Lemma 3.2.14, its reflexive, transitive, symmetric closure is also equivariant. □

Thus, $(A \times X)/{\equiv_\alpha}$ is a nominal set, for any name-set A and nominal set X.

Definition 3.3.20. *The* abstraction of A over X *(written $\langle A\rangle X$) is the quotient nominal set $(A \times X)/{\equiv_\alpha}$. Given terms $a \in A$ and $x \in X$, we write $\langle a\rangle x$ for the $\equiv_\alpha$-equivalence class $[(a, x)]_\alpha$.*

Proposition 3.3.21 (Properties of abstraction). *Let $a \in A$ a name-set and $x \in X$ a nominal set.*

1. *$a, x \mapsto \langle a\rangle x : A, X \to \langle A\rangle X$ is equivariant.*
2. *$supp(\langle a\rangle x) = supp(x) - supp(a) = supp(x) - \{a\}$.*
3. *If $b \in \mathbb{A}$ then $b \# \langle a\rangle x$ if and only if $a = b$ or $b \# x$.*

Proof. Let $a \in A$, $x \in X$ be given.

1. The equivariance of the abstraction-forming operation follows from the fact that

   $$(a\ b) \cdot \langle a'\rangle x = (a\ b) \cdot [(a', x)]_\alpha = [((a\ b) \cdot a', (a\ b) \cdot x)]_\alpha = \langle (a\ b) \cdot a'\rangle (a\ b) \cdot x\ .$$

2. The second equality is obvious.

   To show $supp(\langle a\rangle x) = supp(x) - supp(a)$, first note that $supp(\langle a\rangle x) \subseteq supp(x) \cup supp(a)$. But for any $b \# a, x$ we have $(a\ b) \cdot \langle a\rangle x = \langle b\rangle (a\ b) \cdot x = \langle a\rangle x$, by the definition of equality for abstractions. This shows that $a \notin supp(\langle a\rangle x)$. Hence $supp(\langle a\rangle x) \subseteq supp(x) - supp(a)$. Conversely, suppose $b \in supp(x) - supp(a)$. Then $b \in supp(x)$ but $b \neq a$. Choose $c \# a, x, b$ so that $(b\ c) \cdot x \neq x$. Then

   $$(b\ c) \cdot \langle a\rangle x = \langle (b\ c) \cdot a\rangle (b\ c) \cdot x = \langle a\rangle (b\ c) \cdot x \neq \langle a\rangle x\ .$$

   Hence $b \in supp(\langle a\rangle x)$. This shows $supp(x) - supp(a) \subseteq supp(\langle a\rangle x)$, which completes the proof.

3. This follows immediately from the fact that $supp(\langle a\rangle x) = supp(x) - \{a\}$.

□

Because abstractions are formed using quotient, it can be difficult to work directly with elements of an abstraction, for example in defining a function $f : \langle A\rangle X \to Y$. Instead, it is often more convenient to work in terms of representatives a, x of the equivalence classes $\langle a\rangle x$. The following proposition identifies functions $f : A \times X \times Y \to Z$ that can be "lifted" to functions $f' : \langle A\rangle X \times Y \to Z$.

Proposition 3.3.22. *Let $f : A \times X \times Y \to Z$ be equivariant and assume that whenever $a \# y$, we have $a \# f(a, x, y)$. Then there exists a unique function $f' : \langle A \rangle X \times Y \to Z$ such that $f'(\langle a \rangle x, y) = f(a, x, y)$ for any a, x, y.*

Proof. Let f be as given. We will define a relation R_f as follows: Let $w : \langle A \rangle X$ and y be given. Assume that $a \# y$ and $w = \langle a \rangle x$ for some x. Then $R_f(w, y, f(a, x, y))$ holds.

To see R_f is functional in its first two arguments, suppose $R_f(w, y, f(a, x, y))$ and $R_f(w, y, f(a', x', y))$ both hold, where $a, a' \# y$ and $\langle a \rangle x = w = \langle a' \rangle x'$. So $a \# f(a, x, y)$ and $a' \# f(a', x', y)$. If $a = a'$ then $x = x'$ and we are done. Otherwise, assume $a \# a'$, $a \# x'$ and $x = (a\ a') \cdot x'$. Hence,

$$f(a', x', y) = (a\ a') \cdot f(a', x', y) = f(a, (a\ a') \cdot x', y) = f(a, x, y).$$

This shows that $R_f(w, y, z)$ is functional in w, y.

Define $f'(w, y)$ as the unique z such that $R_f(w, y, z)$ holds. Clearly, f' satisfies $f'(\langle a \rangle x, y) = f(a, x, y)$. This equation specifies the value of f' at every point in its domain so f' is unique. □

3.3.3 The Category of Nominal Sets

$\mathcal{I}$-supported nominal sets and equivariant functions form the objects and arrows of a *category of nominal sets*, which we denote $\mathbf{Nom}(\mathcal{I})$. This category is closed under the formation of finite products, sums, function spaces, and the power set construction. Therefore,

Proposition 3.3.23. $\mathbf{Nom}(\mathcal{I})$ *is a Cartesian closed category with a subobject classifier (i.e., a topos).*

The category of G-sets for a particular G is isomorphic to the functor category $\mathbf{Set}^G$. $\mathbf{Nom}(\mathcal{I})$ is essentially the same as the category of continuous $\mathbb{G}$-sets with respect to some topology on $\mathbb{G}$, which is in turn equivalent to a sheaf category defined in terms of the topology on G. As a special case, when $\mathcal{I} = \mathcal{P}_{<\omega}(\mathbb{A})$, the set of all finite subsets of $\mathbb{A}$, the corresponding sheaf category is the *Schanuel topos* [67]. However, it is not clear just what topological properties $\mathbb{G}$ should have in general in order to assure that minimum supports exist. The presentation we have chosen (using support ideals) seems more natural and direct. But the connection between nominal logic and this corner of category theory deserves further attention.

Note that an equivariant function $f : X \to Y$, viewed as a set of ordered pairs, is an equivariant relation on $X \times Y$, identified by an arrow $p_f : \mathbf{1} \to \mathcal{P}(X \times Y)$. Conversely, an equivariant element of X corresponds to an equivariant function from $\mathbf{1} \to X$, where $\mathbf{1}$ is the terminal object of $\mathbf{Nom}(\mathcal{I})$. Thus, in category-theoretic terms, the equivariant elements of X are so-called *constants* or *global elements*. These observations justify the reuse of the term *equivariant* for functions, relations, and values. It is important to note that many elements of a given nominal set may not be equivariant, and (as witnessed by $\mathbb{A}$), nominal sets may have no equivariant elements at all.

3.4 Nominal Lattices and Fixed Points

In this section we extend some basic results from lattice and fixed-point theory to nominal sets. See [28] for additional background.

In ordinary set theory, sets can be constructed by taking least fixed points of *monotone* set operations (on the power set lattice of some set). A very common example is the construction of the set of terms of a language specified using a BNF grammar. For example, given the λ-calculus grammar

$$e ::= x \mid \lambda x.t \mid t\ t'$$

and a set V of variable names, we can construct a monotone set operator

$$\tau(S) = V \uplus (V \times S) \uplus (S \times S)\ .$$

This function is obviously monotone, and so must have a least and greatest fixed point, by the Knaster-Tarski theorem. In fact, it is also *continuous*, that is, if $S_0 \subseteq S_1 \subseteq \cdots$ is an increasing chain of countably many sets, then

$$\tau(\bigcup_i S_i) = \bigcup_i (\tau(S_i))$$

Therefore, by Kleene's fixed-point theorem, τ's least fixed point is $\tau^\omega(\varnothing)$. That is, its fixed point is reached after at most countably many steps from $\varnothing$. Moreover, $lfp(\tau)$ is essentially the set of λ-terms over variables from V, *not* considered up to α-equivalence.

We consider a more general form of *nominal BNF grammars* which includes syntax for name-sets and name-abstractions. We will regard certain symbols x, y in such a grammar to be *name-symbols* associated with name-sets A_x, A_y, and the notation $x.t, y.e$ will indicate the abstractions $\langle A_x \rangle S_t$, $\langle A_y \rangle S_e$. For example, an appropriate "nominal BNF" grammar for the λ-calculus

$$e ::= x \mid \lambda x.t \mid t\ t'$$

should interpret the x-case as a name-set A and the $\lambda x.t$-case as an abstraction over A:

$$\tau(S) = A_x \uplus \langle A_x \rangle S \uplus (S \times S)$$

In the rest of this section, we will address the question of when such operators have fixed points in nominal sets equipped with lattice structure.

3.4.1 Nominal Fixed Point Theory

By the Knaster-Tarski theorem, a monotone operation on any set with lattice structure (including, for example, a nominal set) has least and greatest fixed points. However, under many circumstances we want to know whether the fixed points are *equivariant*. For example, in a fixed point construction on a nominal powerset

$\mathcal{P}(X)$, we want to know whether $lfp(\tau)$ and $gfp(\tau)$ are nominal subsets of X, that is, whether they are equivariant as elements of $\mathcal{P}(X)$. Clearly, there are trivial monotone operators τ on nominal sets such that $lfp(\tau)$ or $gfp(\tau)$ is not a nominal set, for example $\tau : \mathcal{P}(A) \to \mathcal{P}(A)$ where $\tau(S) = \{a\}$ for some fixed $a \in A$. In this case $lfp(\tau) = gfp(\tau) = \{a\}$ which is not a nominal set since it is not closed under swapping. Obviously, τ is not an equivariant function, and so $\tau(S)$ may not be equivariant even when S is.

On the other hand, if τ is an equivariant, monotone operation on nominal sets, then we can find least and greatest fixed points among the nominal sets. We will show a more general form of this result for an arbitrary nominal set equipped with complete lattice structure compatible with equivariance.

Definition 3.4.1 (Nominal lattice). *A function is* monotone *if $x \leq y$ implies $f(x) \leq f(y)$.*

L is a nominal lattice *provided it has a lattice structure $(|L|, \leq, \wedge, \vee)$ and a nominal-set structure $(|L|, \cdot_L)$ such that $\leq, \wedge, \vee$ are equivariant relations and functions respectively, and $\cdot_L$ is monotone.*

A complete nominal lattice is a nominal lattice that is complete as a lattice, with $\bigwedge$ and $\bigvee$ equivariant.

A (nominal lattice) map *from L to L' is an equivariant, monotone function from L to L'.*

Note that nominal powersets are complete nominal lattices with respect to $\subseteq$, since they are closed under arbitrary union and intersection and inclusion, union, and intersection are equivariant.

We call an element $x \in L$ such that $\tau(x) \leq x$ a *pre-fixed point* of τ, and dually if $x \leq \tau(x)$ then x is a *post-fixed point* of τ. An element x that is both a pre- and post-fixed point is a fixed point, i.e., $x = \tau(x)$.

Theorem 3.4.2 (Nominal Knaster-Tarski). *If L is a complete nominal lattice and $\tau : L \to L$ is a map on L, then*

$$gfp(\tau) := \bigvee\{x \in L \mid x \leq \tau(x)\}$$

and

$$lfp(\tau) := \bigwedge\{x \in L \mid \tau(x) \leq x\}$$

are equivariant elements of L and are the least and greatest fixed points of τ respectively.

Proof. By the ordinary Knaster-Tarski theorem, $lfp(\tau)$ and $gfp(\tau)$ are least and greatest fixed points of τ in L as a lattice, respectively. We must show that they are equivariant with respect to the swapping action inherited from L.

For $lfp(\tau)$, we need to show that $(a\ b) \cdot lfp(\tau) = lfp(\tau)$. Writing $Pre(\tau)$ for the set of pre-fixed points of τ, we need to show that $(a\ b) \cdot \bigvee Pre(\tau) = \bigvee Pre(\tau)$. Since $\bigvee$ is equivariant, it suffices to show that $(a\ b) \cdot Pre(\tau) = Pre(\tau)$, or in particular, that $x \leq \tau(x)$ iff $(a\ b) \cdot x \leq \tau((a\ b) \cdot x)$. This is the case by the equivariance of $\leq$ and τ.

The proof for $gfp(\tau)$ is dual. □

Thus, $lfp(\tau)$ is also the least pre-fixed point, and $gfp(\tau)$ is the greatest post-fixed point of τ.

Definition 3.4.3. *A nominal lattice map $\tau : L \to L$ is* continuous *provided for every ω-chain $x_1 \leq x_2 \leq \cdots$ in L,*

$$\tau(\bigvee_i x_i) = \bigvee_i (\tau(x_i)) .$$

Theorem 3.4.4 (Nominal Kleene Fixed Point). *If $\tau : L \to L$ is a continuous map on the complete nominal lattice L, then $lfp(\tau)$ is equivariant and*

$$lfp(\tau) = \tau^\omega(0)$$

Proof. This follows immediately from the nominal Knaster-Tarski theorem and Kleene's theorem for ordinary lattices. □

Corollary 3.4.5. *If X is a nominal set and $\tau : \mathcal{P}(X) \to \mathcal{P}(X)$ is a continuous map on $\mathcal{P}(X)$ then $lfp(\tau)$ is a nominal set.*

3.4.2 Some Continuous Operations

Many of the standard set-forming operations such as product, sum, and so on are continuous (considered as binary operators on the lattice of sets). Moreover, any set operation built up out of basic sets and such continuous operators is continuous. Thus, a wide variety of inductive constructions can be performed without explicit mention of continuity.

To duplicate this pleasant situation for nominal abstract syntax, we need to check that the basic constructions such as products and sums are continuous; this is straightforward. We also need to check that the operations are *equivariant*; this is sufficient to guarantee that the operations preserve nominal sets. We must do the same for the abstraction construction since we wish to use it to construct nominal abstract syntax. This will be enough to guarantee that operators formed using products, sums, abstractions, and basic nominal sets are continuous maps.

Proposition 3.4.6. *Let A be a name-set. The following constructions are continuous maps in each argument:*

1. $X, Y \mapsto X \times Y$

2. $X, Y \mapsto X + Y$

3. $X \mapsto \langle A \rangle X$,

Moreover, any operation constructed using these operations on nominal sets is continuous.

Proof. All of the transformations are clearly monotone.

The equivariance of all the transformations follows from the fact that in each case the arguments and result are nominal sets, hence closed under swapping. Thus for products we have

$$(a\ b) \cdot (X \times Y) = X \times Y = (a\ b) \cdot X \times Y$$

Sums are similar. For abstractions,

$$(a\ b) \cdot \langle A \rangle X = \langle A \rangle X = \langle A \rangle (a\ b) \cdot X$$

For products and sums, continuity can be proved as for ordinary set product and disjoint union constructions. For abstractions, note that we only consider continuity in the second argument; we will not need continuity in the first name-set argument since name-sets are not allowed to be constructed by induction. We need to show

$$\bigcup_i \langle A \rangle X_i = \langle A \rangle \bigcup_i X_i$$

Let $x \in \bigcup_i \langle A \rangle X_i$ be given. Then for some i, $x \in \langle A \rangle X_i$, so for some $a \in A$ and $y \in X_i$, $x = \langle a \rangle y$. Hence $x = \langle a \rangle y \in \langle A \rangle \bigcup_i X_i$. Conversely, let $x \in \langle A \rangle \bigcup_i X_i$ be given. Then $x = \langle a \rangle y$ for some $a \in A$ and $y \in \bigcup_i X_i$. Hence $y \in X_i$ for some i. So $x = \langle a \rangle y \in \langle A \rangle X_i \subseteq \bigcup_i \langle A \rangle X_i$.

For the last part, equivariance and continuity are preserved under composition and application. □

3.5 Nominal Terms

The primary motivation for nominal sets is in developing a convenient model of abstract syntax with binding, i.e., nominal abstract syntax. We have now developed enough of the theory of nominal sets to show how this can be done. In this section, we define a universe of *nominal terms*, or terms constructed using term symbols, names, and abstractions. We will show that swapping, equality, and freshness can be decided effectively for nominal terms, give induction and recursion principles for them, and define a capture-avoiding substitution operation on nominal terms.

In the next chapter, we will develop nominal logic, a logic codifying some of the behavior of nominal sets. This development will rely on this chapter in two slightly different ways: the *syntax* of nominal logic will be constructed using the universe of nominal terms, and the *semantics* of nominal logic will be developed using abstract nominal sets. In addition, in Chapter 5, the Herbrand models will be constructed using nominal terms.

3.5.1 The Universe of Nominal Terms

Let *Sym* be a nominal set containing (equivariant) constant and function symbols and consider the function

$$\tau(S) = \mathbf{1} \mid Sym \times S \mid S \times S \mid \mathbb{A} \mid \langle\mathbb{A}\rangle t$$

If S is a nominal set, then so is $\tau(S)$. Consequently, $\mathbf{0} \subseteq_{\mathbf{Nom}} \tau(\mathbf{0}) \subseteq_{\mathbf{Nom}} \cdots$ is a chain of nominal sets, so $\bigcup_i \tau^i(\mathbf{0})$ is also a nominal set. We call this nominal set $\mathbb{NT}$, or the set of *nominal terms*. Note that $\mathbb{NT}$ is the least fixed point of τ. We also refer to the subset of $\mathbb{NT}$ constructed using only unit, pairing, and function symbol application as the set of *first-order terms*, or $\mathbb{T}$.

We adopt a more readable notation for elements of $\mathbb{NT}$. We write $\langle\rangle$ for the first case, $f(t)$ for the second case, $\langle t, u\rangle$ for the third, a for the fourth, and $a.t$ for the fifth. Thus, nominal terms may be written according to the nominal BNF grammar

$$t ::= \langle\rangle \mid f(t) \mid \langle t, u\rangle \mid a \mid a.t$$

Alternatively, we may write constant symbols $c\langle\rangle$ as c and function symbol applications $f(\langle t_1, \langle t_2, \ldots, \langle\rangle \cdots\rangle\rangle)$ as $f(\vec{t})$.

3.5.2 Induction

The construction of $\mathbb{NT}$ suggests an induction principle, which we state as follows.

Theorem 3.5.1. *Let X be a nominal set. Suppose P is an equivariant relation on $\mathbb{NT} \times X$, and*

1. *$P(\langle\rangle, x)$ holds,*
2. *$P(f(t), x)$ holds whenever $P(t, x)$ holds,*
3. *$P(\langle t, u\rangle, x)$ holds whenever $P(t, x), P(u, x)$ hold,*
4. *$P(a, x)$ holds whenever $a \in \mathbb{A}$,*
5. *$P(a.t, x)$ holds whenever $a \mathrel{\#} x$ and $P(t, x)$.*

Then $P(t, x)$ holds for all $t \in \mathbb{NT}$.

Proof. Assume that 1–5 hold. We prove that $P(t, x)$ holds for all $t \in \tau^i(0)$ for all i; the desired result follows immediately. Let x be given.

If $i = 0$ then the conclusion is vacuous.

If $i = n + 1$ and $P(t, x)$ holds for all $t \in \tau^n(\mathbf{0})$, then let $t \in \tau^{n+1}(\mathbf{0})$ be given. There are several cases. Those for $\langle\rangle, f(t), \langle t, u\rangle$ are standard, and that for a name a is easy. For the case t an abstraction in $\langle\mathbb{A}\rangle\tau^n(\mathbf{0})$, let a be given and assume $a \mathrel{\#} x$. Assume $t = b.u$ for some name $b \mathrel{\#} x$ and term u. By induction, $P(u, x)$ holds, so $P(b.u, x)$ holds by (5). □

This induction principle can be generalized much further. We will use more general forms of induction over nominal terms without explicit proofs of all the principles.

3.5.3 Recursion

Using the induction principle, we can define functions by (primitive) recursion on nominal terms.

Theorem 3.5.2. *Let X, Y be nominal sets and let $F_f : Sym \times X \times Y \to X$, $F_1 : 1 \times Y \to X$, $F_\times : X \times X \times Y \to X$, $F_\mathbb{A} : \mathbb{A} \times Y \to X$, and $F_{abs} : \mathbb{A} \times X \times Y \to X$ be equivariant functions such that $a \# F_{abs}(a, x, y)$ whenever $a \in \mathbb{A}, x \in X$ and $a \# y$. Then there is a unique function $F : \mathbb{NT} \to X$ such that*

$$\begin{aligned}
F(\langle\rangle, y) &= F_1((), y) && (3.12)\\
F(f(t), y) &= F_f(f, F(t, y), y) && (3.13)\\
F(\langle t, u\rangle, y) &= F_\times(F(t, y), F(u, y), y) && (3.14)\\
F(a, y) &= F_\mathbb{A}(a) && (3.15)\\
F(\langle a\rangle t, y) &= F_{abs}(a, t, y) && (3.16)
\end{aligned}$$

Proof. We define $R_f(t, y, x)$, a relation such that if $F(t, y) = x$ is one of the equations (3.12)–(3.16), then $R_F(t, x)$ is true. We prove that R_F is functional by induction on t, y and define F to be the corresponding function. The only tricky case is for abstraction: there we use Proposition 3.3.22. Any other function satisfying the given equations must be equal to F, as can also be shown by induction on t. □

This recursion principle can be also be generalized much further, and the more general forms will be used without proof.

3.5.4 Computing swapping, equality, and freshness

Using the recursion principle for nominal terms we can define a swapping function as follows.

Definition 3.5.3 (Ground swapping). *We define swapping as follows for ground nominal terms:*

$$\begin{aligned}
\langle\rangle(a\leftrightarrow a') &= \langle\rangle\\
f(t)(a\leftrightarrow a') &= f(t(a\leftrightarrow a'))\\
\langle t, u\rangle(a\leftrightarrow a') &= \langle t(a\leftrightarrow a'), u(a\leftrightarrow a')\rangle\\
b(a\leftrightarrow a') &= (a\ a') \cdot_\mathbb{A} b \quad (b \in \mathbb{A})\\
(b.t)(a\leftrightarrow a') &= (b(a\leftrightarrow a')).(t(a\leftrightarrow a'))
\end{aligned}$$

We must first verify that the abstraction case satisfies the criterion of Theorem 3.5.2.

Lemma 3.5.4. *If $b \# a, a'$ then $b \# (b.t(a\leftrightarrow a')$.*

Proof. Recall that $(b.t)(a, a'\leftrightarrow\ =)(b(a\leftrightarrow a')).(t(a\leftrightarrow a'))$. But since $b \# a, a'$, we have $b(a\leftrightarrow a') = b$, so $b \# b.(t(a\leftrightarrow a')) = b.t(a, a'\leftrightarrow)$. □

Thus, *swap* defines a function on nominal terms. This function correctly and efficiently calculates the result of swapping on two terms:

Proposition 3.5.5. *The swap function can be computed in linear time, and* $t(a\leftrightarrow b) = (a\ b) \cdot_{\mathbb{NT}} t$.

Proof. Obviously $\cdot(a\leftrightarrow b)$ performs a single traversal over the term. That $\cdot(a\leftrightarrow b)$ calculates real swapping can be shown by induction on t. □

In addition, we can define functions calculating whether two nominal terms are equal and whether a name is fresh for a term as follows.

Definition 3.5.6 (Ground freshness). *We compute freshness as follows for ground terms:*

$$\begin{array}{rcl} fresh(a,b) & = & \begin{cases} true & a \neq b \\ false & a = b \end{cases} \\ fresh(a,\langle\rangle) & = & true \\ fresh(a,f(\vec{t})) & = & fresh(a,t_1) \wedge \cdots \wedge fresh(a,t_n) \\ fresh(a,a.t) & = & true \\ fresh(a,b.t) & = & a \neq b \wedge fresh(a,t) \end{array}$$

Definition 3.5.7 (Ground equality). *We define equality as follows for ground terms:*

$$\begin{array}{rcl} eq(a,a) & = & true \\ eq(\langle\rangle,\langle\rangle) & = & true \\ eq(f(t),f(u)) & = & eq(t,u) \\ eq(\langle t_1,u_1\rangle,\langle t_2,u_2\rangle) & = & eq(t_1,u_1) \wedge eq(t_2,u_2) \\ eq(a.t,a.u) & = & eq(t,u) \\ eq(a.t,b.u) & = & a \neq b \wedge fresh(a,u) \wedge eq(t,u(a\leftrightarrow b)) \\ eq(t,u) & = & false \quad otherwise \end{array}$$

These functions correctly decide equality and freshness for nominal terms.

Proposition 3.5.8. *$fresh$ is computable in linear time, and $fresh(a,t) = true$ if and only if $a \# t$.*

Proof. $fresh(a,t)$ performs a single traversal over t, which takes linear time. To show that $fresh$ calculates freshness, the proof is by induction t. The only tricky case is for abstractions, which follows from Proposition 3.3.21(3). □

Proposition 3.5.9. *The eq function is computable in quadratic time. In addition, $eq(t,u) = true$ iff $t = u$.*

Proof. For the complexity bound, the only interesting case is the second one for abstraction. In this case, to solve a problem of size $2n + 2$ we need to solve a freshness problem of size n (taking $O(n)$ time), a swapping problem of size n (taking $O(n)$ time, and an equality problem of size $2n$. If abstractions are nested then we will need to perform work $2n + 2(n-2) + 2(n-4) + \ldots$, which is bounded by $O(n^2)$.

To show that *eq* calculates nominal term equality, it suffices to show this by simultaneous induction on t, u. The only interesting case is for abstraction, but this case follows from the fact that

$$\langle a\rangle t = \langle b\rangle u \iff (a = b \wedge t = u) \vee (a \# u \wedge t = (a\ b) \cdot u)$$

□

3.5.5 Capture-avoiding substitution

Capture-avoiding substitution is an important operation for many applications of abstract syntax. In nominal abstract syntax, substitution is not provided "for free", but it is not hard to define *once and for all* for a nominal term language.

We define a general form of capture-avoiding substitution as follows.

Definition 3.5.10. *Given a nominal term language, we define the* capture-avoiding substitution *of u for v in t by induction on t as:*

$$t\{u/v\} = \begin{cases} u & (t = v) \\ a & (a = t \neq v, a \in \mathbb{A}) \\ \langle\rangle & (\langle\rangle = t \neq v) \\ \langle t_1\{u/v\}, t_2\{u/v\}\rangle & (\langle t_1, t_2\rangle = t \neq v) \\ f(t'\{u/v\}) & (f(t') = t \neq v) \\ \langle a\rangle(t'\{u/v\}) & (\langle a\rangle t' = t \neq v, a \# t, u, v) \end{cases}$$

That this is a function follows from the recursive definition principle for nominal terms. In addition, it is not difficult to verify that, for a sublanguage of nominal terms corresponding to the λ-calculus, that is, for the least subset Λ of $\mathbb{NT}$ satisfying

$$var : V \to \Lambda, app : \Lambda \times \Lambda \to \Lambda, lam : \langle V\rangle\Lambda \to \Lambda$$

we have $\ulcorner t\urcorner\{\ulcorner u\urcorner/var(x)\} = t[u/x]$, where t, u are ordinary λ-terms and $\ulcorner t\urcorner, \ulcorner u\urcorner$ are their representations using nominal terms, and $t[u/x]$ is ordinary capture-avoiding substitution in the λ-calculus, as defined in Chapter 1.

3.6 Notes

The definition of nominal sets in this chapter differs from that of Pitts [108] in only one respect: instead of requiring nominal sets to be finitely supported, nominal sets may contain objects with infinite support as long as supports are small with respect to a support ideal $\mathcal{I}$, and no name-sets are small. This seemingly minor change required redoing much of the theory of nominal sets, but will make it possible to prove completeness for NL in Chapter 5.

Much of the development and many of the proofs of properties of nominal sets in this chapter are based on Gabbay and Pitts' work [42, 39, 43, 108]. The overall development is similar, but the presence of support ideals complicates matters, so this chapter is not simply a restatement of their results.

Pierce [106] and Lambek and Scott [66] are useful sources for basic category theory and the relationship between topos theory and intuitionistic logic; Mac Lane and Moerdijk [67] provides a thorough treatment of topos theory, logic, topology and sheaf theory, and the relationships among them, and in particular the functor category $\mathbf{Set}^G$ which is isomorphic to the category of G-sets. In particular, Pitts' finite-support nominal sets were inspired by the *Schanuel topos*, i.e., the category of continuous G-sets, where the topology on G is generated by the finitely-supported subgroups of G. I have avoided making category theory prerequisite for this chapter. There may well be interesting insights to be gained by taking a more topological or categorical approach to these issues. Menni [76] presents a general, categorical treatment of И-quantification and binding, and Schöpp and Stark [114] have developed a dependent type theory with names and binding based in part on Menni's work and on an analysis of the Schanuel topos.

Chapter 4

Nominal Logic

Logic: The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding.

—Ambrose Bierce

Nominal logic, introduced by Pitts [108], is a first-order theory for reasoning about nominal sets. We introduce a revised form of nominal logic (NL). Our formulation of NL differs from Pitts' in several respects:

- It includes name-constants of name-sort (representing the names inhabiting name-sets) as well as И-quantification over name-constants.
- Its semantics is given in terms of ideal-supported nominal sets, and there is an explicit case for И.
- Its proof system, NL$^{\Rightarrow}$ is a sequent calculus which makes use of name-constants and a structured context to simplify reasoning about И.
- Important properties like completeness and a generalization of Herbrand's theorem hold for our formulation (see Chapter 5).

In this section, we present the syntax, semantics, and a proof system for NL. The issues of soundness, completeness, and Herbrand models will be addressed in Chapter 5.

4.1 Syntax

Let $NSort$, $DSort$ be disjoint, countable sets of *name-sort* and *data-sort* identifiers. The *sorts* of NL are constructed using the following grammar:

$$\sigma ::= \mathbf{1} \mid \delta \mid \nu \mid \langle\nu\rangle\sigma \mid \sigma \times \sigma \tag{4.1}$$

where $\delta \in DSort$, $\nu \in NSort$.

We assume that the set of names $\mathbb{A}$ is partitioned into countable name-sets A^ν, one for each name-sort symbol, inhabited by *name-constants* $\mathsf{a}^\nu, \mathsf{b}^{\nu'}, \ldots$, and countable name-sets V^σ, one for each sort, inhabited by *variables* $x^\sigma, y^{\sigma'}, \ldots$. We write A for the set of all name-constants and V for the set of all variables. Two name-constants or variables of the same sort are said to be *compatible*.

Let *Sym* be a countable set of *term symbols* $c, c', f, f', p, p', \ldots$. The terms of NL are constructed by the following BNF grammar:

$$t, u, a, b ::= \mathsf{a}^\nu \mid x^\sigma \mid f(t) \mid \langle t, t'\rangle \mid \langle\rangle \mid (a\ b) \cdot t \mid \langle a\rangle t \tag{4.2}$$

where $f \in \mathit{Sym}$, $x^\sigma \in V$, $\mathsf{a}^\nu \in \mathbb{A}$, and the notation $\vec{x}$ denotes a sequence $(x_1, \ldots, x_n)$. We typically use the letters a, b for terms of name-sort and t, u for arbitrary terms. There are two built-in function symbols, **swap** and **abs**; we regard the notations $(a\ b) \cdot t$ and $\langle a\rangle t$ as abbreviations for the terms $\mathbf{swap}\langle a, b, t\rangle$ and $\mathbf{abs}\langle a, t\rangle$. Note that the case for abstraction, $\langle a\rangle t$, does not use the dot notation and so abstractions are *not* subject to (meta-level) α-equivalence. Instead, α-equivalence for abstractions will be axiomatized at the object level.

A term $c\langle\rangle$, where $c : \mathbf{1} \to \sigma$, is called a *constant* of σ. We often omit the application to unit for constants, writing just c. If $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$, then we write $f(t_1, \ldots, t_n)$ as an abbreviation for $f(\langle t_1, \langle \cdots, t_n\rangle \cdots\rangle)$.

We write $FV(t)$ and $FN(t)$ for the sets of free variables and name-constants of t, respectively, and use set-theoretic notation $\mathsf{a} \notin FN(t)$, $x \notin FV(t)$, rather than freshness $\mathsf{a} \#_{\mathbb{NT}} t, x \#_{\mathbb{NT}} t$. This is to prevent confusion between the meta-level, where nominal terms are used to take care of the details of abstract syntax, and the object-level, that is, the semantics of nominal logic.

The *formulas* of NL are defined by the following nominal BNF grammar:

$$\begin{array}{rcl} \phi & ::= & \top \mid \bot \mid a \# t \mid t \approx u \mid p(t) \mid \phi \supset \psi \mid \phi \wedge \psi \mid \phi \vee \psi \\ & \mid & \forall x^\sigma.\phi \mid \exists x^\sigma.\phi \mid И\mathsf{a}^\nu.\phi \end{array}$$

where $p \in \mathit{Sym}$. Note here, that $И$, $\exists$, and $\forall$ are identified up to α-equivalence at the meta-level. That is, we view $\forall x.p(x)$ and $\forall y.p(y)$ as the same formula, since they are represented by the same nominal terms (using abstraction for the bound variables.) There are two built-in relation symbols, **eq** and **fresh**. We regard $t \approx u$ and $t \# u$ as abbreviations for $\mathbf{eq}\langle t, u\rangle$ and $\mathbf{fresh}\langle t, u\rangle$. In addition, we take $(a\ b) \cdot \phi$ to be an abbreviation for the formula $\phi((a\ b) \cdot \vec{\mathsf{a}}, (a\ b) \cdot \vec{X})$, where $\vec{\mathsf{a}} = FN(\phi)$ and $\vec{X} = FV\phi$.

We omit the sort-tags on name-constants and variables when clear from context. We take the syntactic name-swapping $t(a {\leftrightarrow} b)$ and capture-avoiding substitution operation $t\{u/x\}$ to be as given in Section 3.5.

Observe that name-constant symbols are regarded as a separate class of meta-level names, distinct from variables and ordinary constants and function symbols. Therefore, the name-constant a^ν is not the same term as the name-sorted variable a^ν. This distinction is important in avoiding the kind of inconsistency that arises in Pitts' system if ground terms of sort ν are present (see [108, Cor. 1, Sec. 5] and Section 5.4).

Remark 4.1.1 (Warning). Note that $(\mathsf{a}\ \mathsf{b}) \cdot \phi$ and $\phi(\mathsf{a}{\leftrightarrow}\mathsf{b})$ are not different notations for the same operation. For example,

$$(\mathsf{a}\ \mathsf{b}) \cdot \text{И}\mathsf{a}.p(\mathsf{a}, x) = \text{И}\mathsf{a}'.p(\mathsf{a}', (\mathsf{a}\ \mathsf{b}) \cdot x)$$

where α-renaming is necessary to propagate the swapping under the quantifier, and the swapping is left suspended on X, whereas

$$(\text{И}\mathsf{a}.p(\mathsf{a}, x))(\mathsf{a}{\leftrightarrow}\mathsf{b}) = \text{И}\mathsf{b}.p(\mathsf{b}, x)\ .$$

In addition, the swapping operator may swap arbitrary (well-formed) terms, for example $((\mathsf{a}\ Y) \cdot X\ (X\ X) \cdot \mathsf{a}) \cdot \phi$ is a legal application of a swapping to a formula. Intuitively, $(a\ b) \cdot x$ is *semantic* swapping of the values of a, b in the value of x, whereas $t(a{\leftrightarrow}b)$ is *syntactic* swapping of the symbols a, b in the term or formula t.

4.1.1 Languages, Contexts, and Well-Formedness

As usual in logic, we need to specify the sort and term languages over which we will operate.

Definition 4.1.2 (Languages). *A* language $\mathcal{L}$ *is a structure*

$$\mathcal{L} = (NSort, DSort, \{f : \sigma \to \sigma', \ldots\}, \{p : \sigma \to \mathbf{prop}, \ldots\})$$

identifying the name- and data-sorts, and listing additional function and relation symbols along with their sorts.

The built-in functions and relation symbols have types

$$\begin{aligned}
\mathbf{swap}_{\nu\sigma} &: \nu \times \nu \times \sigma \to \sigma \\
\mathbf{abs}_{\nu\sigma} &: \nu \times \sigma \to \langle\nu\rangle\sigma \\
\mathbf{eq}_{\sigma} &: \sigma \times \sigma \to \mathbf{prop} \\
\mathbf{fresh}_{\nu\sigma} &: \nu \times \sigma \to \mathbf{prop}
\end{aligned}$$

The subscripts are usually omitted.

In addition, we will use *contexts* to track the uses of variables and name-constants. Contexts will also contain information about freshness.

Definition 4.1.3 (Contexts). *A* context Σ *is a sequence of variables and name-constants as generated by the following grammar:*

$$\Sigma ::= \cdot \mid \Sigma, x^{\sigma} \mid \Sigma \# \mathsf{a}^{\nu}$$

The set of all contexts is called Ctxt.

$$\frac{\mathsf{a}^\nu \in \Sigma}{\Sigma \vdash \mathsf{a} : \nu} \quad \frac{x^\sigma \in \Sigma}{\Sigma \vdash x : \sigma} \quad \frac{}{\Sigma \vdash \langle\rangle : \mathbf{1}} \quad \frac{f : \sigma \to \sigma' \in \mathcal{L} \quad \Sigma \vdash t : \sigma}{\Sigma \vdash f(t) : \sigma'}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1, t_2 : \sigma_2}{\Sigma \vdash \langle t_1, t_2\rangle : \sigma_1 \times \sigma_2} \quad \frac{\Sigma \vdash a, b : \nu, t : \sigma}{\Sigma \vdash (a\ b) \boldsymbol{\cdot} t : \sigma} \quad \frac{\Sigma \vdash a : \nu, t : \sigma}{\Sigma \vdash \langle a\rangle t : \langle\nu\rangle\sigma}$$

Figure 4.1: Well-formedness rules for terms

$$\frac{}{\Sigma \vdash \top, \bot : \mathbf{prop}} \qquad \frac{p : \sigma \to \mathbf{prop} \in \mathcal{L} \quad \Sigma \vdash t : \sigma}{\Sigma \vdash p(t) : \mathbf{prop}}$$

$$\frac{\Sigma \vdash \phi : \mathbf{prop}}{\Sigma \vdash \neg\phi : \mathbf{prop}} \qquad \frac{\Sigma \vdash \phi : \mathbf{prop}, \psi : \mathbf{prop}}{\Sigma \vdash \phi \wedge \psi, \phi \vee \psi, \phi \supset \psi : \mathbf{prop}}$$

$$\frac{\Sigma, x^\sigma \vdash \phi : \mathbf{prop}}{\Sigma \vdash \forall x^\sigma.\phi, \exists x^\sigma.\phi : \mathbf{prop}} \qquad \frac{\Sigma \# \mathsf{a}^\nu \vdash \phi : \mathbf{prop}}{\Sigma \vdash \text{И}\mathsf{a}^\nu.\phi : \mathbf{prop}}$$

Figure 4.2: Well-formedness rules for formulas

We write $\mathsf{a}^\nu \in \Sigma$, $x^\sigma \in \Sigma$ if a^ν or x^σ appears in Σ respectively. We write $\Sigma; \Sigma'$ for the sequential composition of two contexts, i.e.,

$$\begin{aligned} \Sigma; \cdot &= \Sigma \\ \Sigma; \Sigma'\#\mathsf{a} &= (\Sigma; \Sigma')\#\mathsf{a} \\ \Sigma; \Sigma', x &= (\Sigma; \Sigma'), x \end{aligned}$$

Let $\mathcal{L}$ be a language and Σ a context. A term t is *well-formed (at sort σ)* provided $\Sigma \vdash t : \sigma$ is derivable using the typechecking rules of Figure 4.1. A formula ϕ is well-formed provided $\Sigma \vdash \phi : \mathbf{prop}$ is derivable using the typechecking rules of Figure 4.2. (The rules with multiple conclusions abbreviate several rules each with the same hypotheses and different conclusions.) We write $\mathbb{T}_\Sigma(\sigma)$ and $\mathbb{F}_\Sigma$ for the sets of terms of sort σ and formulas well-formed with respect to context Σ. Note that the set *Ctxt* of all contexts, $\mathbb{T}_\varnothing(\sigma)$, and $\mathbb{F}_\varnothing$ are each nominal subsets of the term universe $\mathbb{T}$. In what follows, all formulas and terms are assumed to be well-formed with respect to some language and context.

We identify collections of αProlog declarations with languages in the obvious way.

Example 4.1.4. Consider the αProlog signature

$$n : \mathbf{type}. \quad z : n. \quad s : n \to n. \quad iszero(n). \tag{4.3}$$

for unary arithmetic. Well-formed terms and formulas include $iszero(z)$, $s(z) \# z$, and $s(s(z))$.

Example 4.1.5. Consider the αProlog signature

$$\begin{gathered} id : \mathbf{name\_type}. \quad exp : \mathbf{type}. \\ var : id \to exp. \quad app : exp \times exp \to exp. \quad lam : \langle id\rangle exp \to exp \end{gathered} \tag{4.4}$$

for the syntax of λ-terms. Well-formed terms and formulas include $\mathsf{a} \# \mathit{app}(x, y)$, $\mathit{var}(\mathsf{a})$, and $\mathit{lam}(\langle\mathsf{a}\rangle x)$.

Lemma 4.1.6. *If $\Sigma, x : \sigma, \Sigma' \vdash u : \tau$ and $\Sigma \vdash t : \sigma$ then $\Sigma, \Sigma' \vdash u\{t/x\} : \tau$.*
If $\Sigma, x : \sigma, \Sigma' \vdash \phi : \mathbf{prop}$ and $\Sigma \vdash t : \sigma$ then $\Sigma, \Sigma' \vdash \phi\{t/x\} : \mathbf{prop}$.
If $\Sigma \vdash t : \sigma$ then $\Sigma(a\leftrightarrow b) \vdash t(a\leftrightarrow b) : \sigma$.
If $\Sigma \vdash \phi : \mathbf{prop}$ then $\Sigma(a\leftrightarrow b) \vdash \phi(a\leftrightarrow b) : \mathbf{prop}$.

Proof. The proof in each case is by induction on the first derivation, and is easy. □

Note that if a follows x in Σ, then we may not substitute a term containing a for x. The importance of this will be clarified later, when we show how to read freshness constraints from contexts.

4.2 Semantics

In this section, let a fixed language $\mathcal{L}$ be given. We will show how to interpret the nominal language $\mathcal{L}$ in a collection of nominal sets $\mathbf{Nom}^*$ with set of names $\mathbb{A}^*$, name-group $\mathbb{G}^*$, and support ideal $\mathcal{I}^*$ over $\mathbb{A}^*$. We require $\mathbf{Nom}^*$ to contain a terminal element $\mathbf{1}^*$, and to be closed under products and abstraction.

Definition 4.2.1 (Sort interpretations). *A* basic sort interpretation *is a pair of mappings $\delta \mapsto \mathcal{S}[\![\delta]\!] : DSort \to \mathbf{Nom}^*$ from data sorts to nominal sets in $\mathbf{Nom}^*$, and $\nu \mapsto \mathcal{S}[\![\nu]\!] : NSort \to \{A \in \mathbf{Nom}^* \mid A \text{ a name-set}\}$, an injective mapping from name-sorts to name-sets. It can be extended to a full sort interpretation as follows:*

- $\mathcal{S}[\![\mathbf{1}]\!] = \mathbf{1}$.
- *For any name-sort ν and sort σ, $\mathcal{S}[\![\langle\nu\rangle\sigma]\!] = \langle\mathcal{S}[\![\nu]\!]\rangle\mathcal{S}[\![\sigma]\!]$.*
- *For any sorts σ, σ', $\mathcal{S}[\![\sigma \times \sigma']\!] = \mathcal{S}[\![\sigma]\!] \times \mathcal{S}[\![\sigma']\!]$.*

The universe *M of a sort interpretation is the union*

$$M = \bigcup_{\sigma} \mathcal{S}[\![\sigma]\!]$$

of all the interpretations of all the sorts.

Definition 4.2.2 (Valuations). *A* valuation *is a finite-domain partial function $\rho : \mathbb{A} \to M$ satisfying $\rho(x^\sigma) \in \mathcal{S}[\![\sigma]\!]$ for each $x^\sigma \in V$ and $\rho(\mathsf{a}^\nu) \in \mathcal{S}[\![\nu]\!]$ for each $\mathsf{a}^\nu \in \mathsf{A}$.*

A valuation ρ matches *a context Σ (written $\rho : \Sigma$) if $FV(\Sigma) \subset Dom(\rho)$ and for every sub-context of Σ of the form $\Sigma'\#\mathsf{a}$, we have $\rho(\mathsf{a}) \# \rho(x)$ for every $x \in \Sigma'$.*

The set of all valuations (matching a context Σ) is called Val ($Val(\Sigma)$).

We write $\rho[x \mapsto v]$ for the valuation that maps x to v and otherwise simulates ρ.

Example 4.2.3. Consider the valuation $[x \mapsto \mathsf{a}, y \mapsto \mathsf{b}]$. This is well-formed with respect to the contexts x, y; y, x; a, x, y; and $\mathsf{a}, x\#\mathsf{b}, y$, and so on, but not with respect to $x, y\#\mathsf{a}$ or $x\#\mathsf{a}, y\#\mathsf{b}$, etc.

Remark 4.2.4 (Nominal set of valuations). Note that the set of all finite-domain valuations Val can be viewed as a nominal set, namely, $Val = \bigcap_\sigma V^\sigma \to \mathcal{S}[\![\sigma]\!]$. In addition, the valuation update operation $\rho, x, v \mapsto \rho[x \mapsto v] : Val \times V^\sigma \times \mathcal{S}[\![\sigma]\!] \to Val$ is equivariant. Also, the construction $Val(\Sigma) = \{\rho \in Val \mid \rho : \Sigma\}$ is equivariant, so $supp(Val(\Sigma)) = supp(\Sigma)$.

Definition 4.2.5 (Term interpretations). *A* basic term interpretation *is a mapping $f \mapsto \mathcal{T}[\![f]\!]$ such that if $\Sigma \vdash f : \sigma \to \sigma'$, then $\mathcal{T}[\![f]\!]$ is an equivariant function mapping $\mathcal{S}[\![\sigma]\!] \to \mathcal{S}[\![\sigma']\!]$.*

If $\Sigma \vdash t : \sigma$ and $\rho : \Sigma$ then the value $\mathcal{T}[\![t]\!]\rho$ of t is defined by the rules

$$\begin{aligned}
\mathcal{T}[\![\mathsf{a}]\!]\rho &= \rho(\mathsf{a}) && (4.5)\\
\mathcal{T}[\![x]\!]\rho &= \rho(x) && (4.6)\\
\mathcal{T}[\![\langle\rangle]\!]\rho &= \langle\rangle && (4.7)\\
\mathcal{T}[\![f(t)]\!]\rho &= \mathcal{T}[\![f]\!](\mathcal{T}[\![t]\!]\rho) && (4.8)\\
\mathcal{T}[\![\langle t_1, t_2\rangle]\!]\rho &= (\mathcal{T}[\![t_1]\!]\rho, \mathcal{T}[\![t_2]\!]\rho) && (4.9)\\
\mathcal{T}[\![(t_1\ t_2) \cdot t]\!]\rho &= (\mathcal{T}[\![t_1]\!]\rho\ \mathcal{T}[\![t_2]\!]\rho) \cdot_{\mathcal{S}[\![\sigma]\!]} \mathcal{T}[\![t]\!]\rho \qquad (t : \sigma) && (4.10)\\
\mathcal{T}[\![\langle t_1\rangle t_2]\!]\rho &= \langle\mathcal{T}[\![t_1]\!]\rho\rangle\mathcal{T}[\![t_2]\!]\rho && (4.11)
\end{aligned}$$

Proposition 4.2.6. *If $\rho : \Sigma$ and $\Sigma \vdash t : \sigma$ then $\mathcal{T}[\![t]\!]\rho \in \mathcal{S}[\![\sigma]\!]$.*

The proof is straightforward.

Definition 4.2.7 (Formula interpretations). *A* basic formula interpretation *is a mapping from uninterpreted predicate symbols $p : \sigma \to$ **prop** to an equivariant relation $\mathcal{P}[\![p]\!]$ on $\mathcal{S}[\![\sigma]\!]$ (or, a nominal subset of $\mathcal{S}[\![\sigma]\!]$). A basic formula interpretation is extended to $\approx$ and $\#$ formulas as follows:*

$$\begin{aligned}
\mathcal{P}[\![\mathbf{fresh}_{\nu\sigma}]\!] &= \{(t, u) \mid t \in \mathcal{S}[\![\nu]\!], u \in \mathcal{S}[\![\sigma]\!], t \# u\} && (4.12)\\
\mathcal{P}[\![\mathbf{eq}_\sigma]\!] &= \{(t, u) \mid t = u \in \mathcal{S}[\![\sigma]\!]\} && (4.13)
\end{aligned}$$

Definition 4.2.8 (Structures). *A* structure $\mathcal{M} = (M, \mathcal{S}[\![\cdot]\!], \mathcal{T}[\![\cdot]\!], \mathcal{P}[\![\cdot]\!])$ *consists of a universe, a sort interpretation, a term interpretation, and a formula interpretation.*

The satisfiability relation $\vDash^{\mathcal{M}}$ relating contexts, $\mathcal{M}$-valuations and formulas is defined by the rules in Figure 4.3. The judgment $\Sigma : \rho \vDash^{\mathcal{M}} \phi$ is well-formed when $\rho : \Sigma$ and $\Sigma \vdash \phi :$ **prop**.

$$\begin{array}{lcll}
\Sigma : \rho & \models & \top & \text{for any } \rho : \Sigma \\
\Sigma : \rho & \not\models & \bot & \text{for any } \rho : \Sigma \\
\Sigma : \rho & \models & \neg\phi & \text{if } \Sigma : \rho \not\models \phi \\
\Sigma : \rho & \models & \phi \wedge \psi & \text{if } \Sigma : \rho \models \phi \text{ and } \Sigma : \rho \models \psi \\
\Sigma : \rho & \models & \phi \vee \psi & \text{if } \Sigma : \rho \models \phi \text{ or } \Sigma : \rho \models \psi \\
\Sigma : \rho & \models & \phi \supset \psi & \text{if } \Sigma : \rho \models \psi \text{ whenever } \Sigma : \rho \models \phi \\
\Sigma : \rho & \models & p(t) & \text{if } \rho(t) \in [\![p]\!] \\
\Sigma : \rho & \models & \forall x^\sigma.\phi & \text{if } \Sigma, x : \rho[x \mapsto v] \models \phi \text{ for all } v \in [\![\sigma]\!],\ x \notin \Sigma \\
\Sigma : \rho & \models & \exists x^\sigma.\phi & \text{if } \Sigma, x : \rho[x \mapsto v] \models \phi \text{ for some } v \in [\![\sigma]\!],\ x \notin \Sigma \\
\Sigma : \rho & \models & И\mathsf{a}^\nu.\phi & \text{if } \Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \models \phi \text{ for } v \in [\![\nu]\!] - \mathit{supp}(\rho),\ \mathsf{a} \notin \Sigma
\end{array}$$

Figure 4.3: Satisfiability relation for nominal logic

Remark 4.2.9 (Abuse of notation). When there is no possibility of confusion, we write $[\![\cdot]\!]$ for $\mathcal{S}[\![\cdot]\!], \mathcal{T}[\![\cdot]\!], \mathcal{P}[\![\cdot]\!]$, as appropriate; we also write $\rho(t)$ for $\mathcal{T}[\![t]\!]\rho$ when the term interpretation is clear from context. The superscript on $\models^{\mathcal{M}}$ is omitted when clear from context.

Definition 4.2.10 (Models). *We say $\mathcal{M}$* is a model of *ϕ or ϕ* is valid in *$\mathcal{M}$ ($\mathcal{M} \models \phi$) if $\Sigma : \rho \models^{\mathcal{M}} \phi$ for each $\Sigma \vdash \phi :$ **prop** and $\rho : \Sigma$, and $\mathcal{M}$* is a model of *Γ if $\mathcal{M} \models \phi$ for every $\phi \in \Gamma$.*

4.2.1 Examples

Example 4.2.11 (Definable sets of names). Consider a unary predicate symbol $p : \nu \to$ **prop** over a name-sort ν. Because of equivariance, $p(x)$ is either true of every name or of no names. Therefore, there are only two unary predicates on a name-sort, equivalent to truth and falsity, and only two definable subsets of $[\![\nu]\!]$, namely $[\![\nu]\!]$ and $\varnothing$.

Example 4.2.12 (Definable relations on names). Now consider a binary predicate $q : \nu \times \nu \to$ **prop**. By equivariance, the behavior of q is determined by the two formulas $Q_1 = И\mathsf{a}.q(\mathsf{a},\mathsf{a})$ and $Q_2 = И\mathsf{a}.И\mathsf{b}.q(\mathsf{a},\mathsf{b})$. Hence, there are four possible binary relations on names, equivalent to truth, falsity, equality, and freshness, according as both, neither, Q_1 only, and Q_2 only hold for q respectively.

This implies that there can be no nontrivial, equivariant partial ordering on names, that is, no reflexive, transitive, antisymmetric relation $\leq$. For if $\mathsf{a} \leq \mathsf{b}$ holds for $\mathsf{a} \neq \mathsf{b}$ then so does $\mathsf{b} \leq \mathsf{a}$ by equivariance, and we have $\mathsf{a} \approx \mathsf{b}$, contradicting the antisymmetry property of $\leq$.

Example 4.2.13 (Theorems). We now consider some valid formulas.

1. $И\mathsf{a},\mathsf{b}.p(\mathsf{a},\mathsf{b}) \supset p(\mathsf{b},\mathsf{a})$. This reduces to checking that $\mathsf{a}\#\mathsf{b} : \rho \models p(\mathsf{a},\mathsf{b})$ implies $\mathsf{a}\#\mathsf{b} : \rho \models p(\mathsf{b},\mathsf{a})$, which follows from the equivariance of $[\![p]\!]$.

2. $\text{И}\mathsf{a}.\text{И}\mathsf{b}.\mathsf{a} \# \mathsf{b}$. For any valuation $\rho : \mathsf{a} \# \mathsf{b}$, we have $\mathsf{a}\#\mathsf{b} : \rho \vDash \mathsf{a} \# \mathsf{b}$ since $\mathsf{a} \neq \mathsf{b}$.

3. $\forall x.\text{И}\mathsf{a}, \mathsf{b}.(\mathsf{a}\ \mathsf{b}) \cdot x \approx x$. Applying the rules for $\vDash$, this asks whether for any $v \in [\![\sigma]\!]$, $\mathsf{a} \# v$, and $\mathsf{b} \# \mathsf{a}, v$, we have $(\mathsf{a}\ \mathsf{b}) \cdot v = v$. This is evident from Proposition 3.2.16.

4. $\forall x.\text{И}\mathsf{a}, \mathsf{b}.\langle a \rangle x \approx \langle b \rangle x$. This asks whether for any $v \in [\![\sigma]\!]$ and $\mathsf{a} \# \mathsf{b} \# [x \mapsto v]$, we have $\langle \mathsf{a} \rangle x = \langle \mathsf{b} \rangle x$, i.e.

$$(\mathsf{a}, v) \equiv_\alpha (\mathsf{b}, v) \ .$$

Since $\mathsf{a} \neq \mathsf{b}$, the only way that this could be is if $[x \mapsto v] \vDash x \approx (\mathsf{a}\ \mathsf{b}) \cdot x$ and $[x \mapsto v] \vDash \mathsf{a} \# x$; both are clearly true because $\mathsf{a}, \mathsf{b} \# v = [x \mapsto v](x)$.

Example 4.2.14 (Non-theorems). Now we consider some invalid formulas.

1. $\forall a, x, y.a \# f(x, y) \supset a \# x \wedge a \# y$. This formula has a countermodel given by taking $f(x, y) = 0$ for some constant 0 and taking $x, y, a = \mathsf{a}$. Then $\mathsf{a} \# 0 = f(\mathsf{a}, \mathsf{a})$ but $\mathsf{a} \# \mathsf{a}$ is false.

2. $\text{И}\mathsf{a}.\phi(\mathsf{a}, \mathsf{a}) \supset \text{И}\mathsf{a}.\text{И}\mathsf{b}.\phi(\mathsf{a}, \mathsf{b})$. This formula has a counterexample given by taking $\phi(x, y)$ to be $x \approx y$. Then $\text{И}\mathsf{a}.\mathsf{a} \approx \mathsf{a}$ holds but $\text{И}\mathsf{a}, \mathsf{b}.\mathsf{a} \approx \mathsf{b}$ does not.

3. $\text{И}\mathsf{a}.\text{И}\mathsf{b}.\phi(\mathsf{a}, \mathsf{b}) \supset \text{И}\mathsf{a}.\phi(\mathsf{a}, \mathsf{a})$. This formula has a counterexample given by taking $\phi(x, y) = x \# y$. Then $\text{И}\mathsf{a}, \mathsf{b}.\mathsf{a} \# \mathsf{b}$ is valid but $\text{И}\mathsf{a}.\mathsf{a} \# \mathsf{a}$ is not.

4.2.2 Semantic Properties

First we show that nominal models in general satisfy the freshness and equivariance principles. Freshness means that for any finite collection of values we can find a name fresh for all of the values simultaneously.

Theorem 4.2.15 (Logical freshness). *The formula $\forall x^\sigma.\exists a^\nu.a \# x$ is valid in any nominal structure.*

Proof. Let $v \in [\![\sigma]\!]$ be given. Then $S = \mathit{supp}_{[\![\sigma]\!]}(v) \in \mathcal{I}$. Since $\mathsf{A}^\nu \notin \mathcal{I}$, choose $\mathsf{a} \in \mathsf{A}^\nu - S$. Then $[x \mapsto v, a \mapsto \mathsf{a}] \vDash a \# x$ because $\mathsf{a} \notin \mathit{supp}(v)$. The choice of v was arbitrary, so $\mathcal{M} \vDash \forall x.\exists a.a \# x$. □

Note that this property subsumes Pitts' freshness axiom $F4$, which states that $\forall \vec{x}.\exists a.a \# \vec{x}$, because any finite sequence $\vec{x}$ can be translated to a n-tuple $\langle x_1, \ldots, x_n \rangle$ such that $a \# x_1 \wedge \cdots \wedge a \# x_n \iff a \# \langle x_1, \ldots, x_n \rangle$.

Lemma 4.2.16 (Swapping and validity). *Let ρ be a valuation and $\Sigma \vdash a, b : \nu$. The following are equivalent:*

1. $\Sigma : \rho \vDash \phi$

2. $\Sigma : \rho \vDash (a\ a) \cdot \phi$

3. $\Sigma : \rho \vDash (a\ b) \cdot (a\ b) \cdot \phi$.

Proof. The proof is by induction on the structure of ϕ using the definition of $(a\ b) \cdot \phi$. All the cases are straightforward. □

Theorem 4.2.17 (Logical equivariance). *Let ϕ be a formula and a, b terms of some name-sort ν. Then if $\Sigma : \rho \vDash \phi$ then $\Sigma : \rho \vDash (a\ b) \cdot \phi$.*

Proof. Proof is by induction on the structure of ϕ.

- If $\Sigma : \rho \vDash p(t)$ (where p may be either $\approx$, $\#$, or some other relation symbol) then $\rho(t) \in [\![p]\!]$. Set $v = \rho(a), v' = \rho(b)$. Then

  $$\rho((a\ b) \cdot t) = (v\ v') \cdot \rho(t) \in [\![p]\!]$$

  by the equivariance of $[\![p]\!]$ (see Lemma 3.2.14), so $\Sigma : \rho \vDash (a\ b) \cdot p(t)$.

- If $\Sigma : \rho \vDash \top$ then $\Sigma : \rho \vDash (a\ b) \cdot \top = \top$ trivially.

- Since $\Sigma : \rho \nvDash \bot$, the case for $\bot$ is vacuous.

- If $\Sigma : \rho \vDash \neg\phi$ then $\Sigma : \rho \nvDash \phi$ by Lemma 4.2.16. So $\Sigma : \rho \nvDash (a\ b) \cdot (a\ b) \cdot \phi$. By the contrapositive of the induction hypothesis, we must have $\Sigma : \rho \nvDash (a\ b) \cdot \phi$.

- If $\Sigma : \rho \vDash \phi \wedge \psi$ then $\Sigma : \rho \vDash \phi$ and $\Sigma : \rho \vDash \psi$. By induction, then, $\Sigma : \rho \vDash (a\ b) \cdot \phi$ and $\Sigma : \rho \vDash (a\ b) \cdot \psi$ so $\Sigma : \rho \vDash (a\ b) \cdot (\phi \wedge \psi)$. The case for $\vee$ is dual.

- If $\Sigma : \rho \vDash \phi \supset \psi$ then whenever $\Sigma : \rho \vDash \phi$ it follows that $\Sigma : \rho \vDash \psi$. By induction, then, if $\Sigma : \rho \vDash (a\ b) \cdot \phi$ then $\Sigma : \rho \vDash (a\ b) \cdot (a\ b) \cdot \phi$, and then by Lemma 4.2.16, $\Sigma : \rho \vDash \phi$, so $\Sigma : \rho \vDash \psi$. By induction again, $\Sigma : \rho \vDash (a\ b) \cdot \psi$ so $\Sigma : \rho \vDash (a\ b) \cdot (\phi \supset \psi)$.

- If $\Sigma : \rho \vDash \forall x^\sigma.\phi$ then $\Sigma, x : \rho[x \mapsto v] \vDash \phi$ for all $v \in [\![\sigma]\!]$. Then $\Sigma, x : \rho[x \mapsto v] \vDash (a\ b) \cdot \phi$ also for each $v \in [\![\sigma]\!]$. So $\Sigma : \rho \vDash \forall x.(a\ b) \cdot \phi = (a\ b) \cdot \forall x.\phi\{(a\ b) \cdot (a\ b) \cdot x/x\}$, and since $(a\ b) \cdot (a\ b) \cdot x \approx x$, we have $\Sigma : \rho \vDash (a\ b) \cdot \forall x.\phi$. The case for $\exists$ is dual.

- If $\Sigma : \rho \vDash$ И$\mathsf{a}^\nu.\phi$ then $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \phi$ for $\mathsf{a}^\nu \notin \Sigma, v \in [\![\nu]\!] - supp(\rho)$. Obviously $\rho[\mathsf{a} \mapsto v] : \Sigma\#\mathsf{a}$, so by induction we have $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash (a\ b) \cdot \phi$, and we may conclude that $\Sigma : \rho \vDash$ И$\mathsf{a}.(a\ b) \cdot \phi = (a\ b) \cdot$ И$\mathsf{a}.\phi$.

This concludes the proof. □

We now show some important properties of the satisfiability relation which will be needed later on. First, satisfiability is preserved up to changing the context to another one compatible with ρ.

Lemma 4.2.18 (Context change). *If $\Sigma' \vdash \phi : \mathbf{prop}$ and $\rho : \Sigma'$ and $\Sigma : \rho \vDash \phi$ then $\Sigma' : \rho \vDash \phi$.*

Proof. By induction on the structure of ϕ. The cases for propositional and atomic formulas are straightforward. The quantifier cases are more interesting because Σ' may mention different names than Σ. We do the $\forall$ and И cases as representative examples.

If $\phi = \forall x.\psi$, then we know $\Sigma, x : \rho[x \mapsto v] \vDash \psi$ for all $v \in [\![\sigma]\!]$. Without loss of generality, assume x is fresh for Σ' as well as Σ. Let $v \in [\![\sigma]\!]$ be given. Then $\rho[x \mapsto v] : \Sigma', x$ since $\rho : \Sigma$. Then by induction, $\Sigma', x : \rho[x \mapsto v] \vDash \psi$, so $\Sigma' : \rho \vDash \forall x.\psi$.

If $\phi = \text{И}\mathsf{a}.\psi$, then we know $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \psi$ where $\mathsf{a} \notin \Sigma$ and $v \in [\![\nu]\!] - \mathit{supp}(\rho)$. Without loss of generality, assume a is fresh for Σ' as well. Then $\rho[\mathsf{a} \mapsto v] : \Sigma'\#\mathsf{a}$. By induction, $\Sigma'\#\mathsf{a} : \rho \vDash \psi$, so $\Sigma' : \rho \vDash \text{И}\mathsf{a}.\psi$. □

Theorem 4.2.19 (Semantic freshness). *If a^ν is fresh for Σ, ϕ, then $\Sigma : \rho \vDash \phi$ iff $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \phi$ for $v \in [\![\nu]\!] - \mathit{supp}(\rho)$.*

If x^σ is fresh for Σ, ϕ, ρ, then $\Sigma : \rho \vDash \phi$ iff $\Sigma, x : \rho[x \mapsto v] \vDash \phi$ for any $v \in [\![\sigma]\!]$.

Proof. For the forward direction, apply Lemma 4.2.18 with $\Sigma' = \Sigma\#\mathsf{a}$ and $\rho' = \rho[\mathsf{a} \mapsto v]$ for some $v \in [\![\nu]\!] - \mathit{supp}(\rho)$. Clearly $\Sigma' \vdash \phi : \mathbf{prop}$, and since $\rho : \Sigma$ and $\mathsf{a} \# \rho$, we have $\rho : \Sigma'$, so we must have $\Sigma\#\mathsf{a} : \rho \vDash \phi$.

For the reverse direction, the result follows from Lemma 4.2.18 and the facts that if $\rho[\mathsf{a} \mapsto v] : \Sigma\#\mathsf{a}$ then $\rho : \Sigma$ and if $\Sigma\#\mathsf{a} \vdash \phi : \mathbf{prop}$ and a does not appear in ϕ then $\Sigma \vdash \phi : \mathbf{prop}$.

The second part is straightforward. □

Theorem 4.2.20 (Semantic equivariance). *Let a, b be compatible names. Then $\Sigma : \rho \vDash \phi$ if and only if $\Sigma(a \leftrightarrow b) : \rho(a \leftrightarrow b) \vDash \phi(a \leftrightarrow b)$.*

Proof. The proof is a straightforward induction on the structure of ϕ. The only difficult cases are those involving quantifiers, but we can assume that the bound names involved are fresh for a and b so there is no problem. For example, in the $\forall$ case, the proof goes as follows.

Suppose $\Sigma : \rho \vDash \forall x.\psi$. Assume without loss that x is fresh for a, b, Σ, ρ. Then $\Sigma, x : \rho[x \mapsto v] \vDash \psi$ for every $v \in [\![\sigma]\!]$. Let $v \in [\![\sigma]\!]$ be given. Then $\Sigma, x : \rho[x \mapsto v(a \leftrightarrow b)] \vDash \psi$ holds. By induction, we have

$$(\Sigma, x)(a \leftrightarrow b) : (\rho[x \mapsto v(a \leftrightarrow b)])(a \leftrightarrow b) \vDash \psi(a \leftrightarrow b)$$

Moreover, using various equivariance properties, this is equivalent to

$$\Sigma(a \leftrightarrow b), x(a \leftrightarrow b) : \rho(a \leftrightarrow b)[x(a \leftrightarrow b) \mapsto v(a \leftrightarrow b)(a \leftrightarrow b)] \vDash \psi(a \leftrightarrow b)$$

Because $x \# a, b$, we have $x(a \leftrightarrow b) = x$, and because $(a \leftrightarrow b)$ is an involution, we have $v(a \leftrightarrow b)(a \leftrightarrow b) = v$, so

$$\Sigma(a \leftrightarrow b), x : \rho(a \leftrightarrow b)[x \mapsto v] \vDash \psi(a \leftrightarrow b)$$

Since v was arbitrary, we may conclude that $\Sigma(a \leftrightarrow b) : \rho(a \leftrightarrow b) \vDash \phi(a \leftrightarrow b)$. □

4.3 A Sequent Calculus

So far we have established validity and invalidity for several formulas of nominal logic semantically. In this section we introduce a Gentzen-style sequent calculus for NL. This system is an extension of a standard sequent calculus for classical first-order logic.

The judgments are *sequents* of the form $\Sigma : \Gamma \Rightarrow \Delta$, where Σ is a context, and Γ and Δ are finite (multi)sets of formulas well-formed with respect to Σ. Intuitively, a sequent $\Sigma : \Gamma \Rightarrow \Delta$ means "Assuming all the formulas of Γ are true, at least one of the formulas of Δ is true".

The standard sequent calculus rules for of first-order logic are presented in Figure 4.4. The additional rules of NL are shown in Figure 4.6 and Figure 4.7. The notation $\Sigma|_{\mathsf{a}}$ in the $\Sigma\#$ rule is defined in Section 4.3.4. Some convenient admissible rules are shown in Figure 4.5. These rules will be proved to be admissible in Section 4.4; that is, they can be simulated using the base rules of $NL^{\Rightarrow}$.

4.3.1 Basic Properties

The sequent calculus for NL satisfies the following basic properties.

Lemma 4.3.1 (Weakening). *If $\Sigma : \Gamma \Rightarrow \Delta$ is derivable then $\Sigma : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'$ has a derivation of the same height.*

Lemma 4.3.2 (Contraction). *If $\Sigma : \Gamma, \Gamma', \Gamma' \Rightarrow \Delta, \Delta', \Delta'$ is derivable then $\Sigma : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'$ has a derivation of the same height.*

Lemma 4.3.3 (Substitution). *If $\Sigma, x^\sigma; \Sigma' : \Gamma \Rightarrow \Delta$ is derivable and $\Sigma \vdash t : \sigma$ then $\Sigma; \Sigma' : \Gamma\{t/x^\sigma\} \Rightarrow \Delta\{t/x^\sigma\}$, has a derivation of the same height.*

Lemma 4.3.4 (Exchange). *If $\Sigma : \Gamma \Rightarrow \Delta$ is derivable and a and b are compatible then $\Sigma(a{\leftrightarrow}b) : \Gamma(a{\leftrightarrow}b) \Rightarrow \Delta(a{\leftrightarrow}b)$ has a derivation of the same height.*

The proofs of these properties are by straightforward structural induction on derivations.

4.3.2 The Non-Logical Rules

The rules dealing with equality, freshness, swapping, and abstraction are referred to as *non-logical rules* because they deal with atomic formulas rather than logical connectives. In this section we give some intuition for the meanings and purposes of these rules.

- (S_1)–(S_3): These axioms describe basic properties of *swapping*. From them, additional properties are derivable, for example using (S_2) and (S_3):

  $$(a\ b) \cdot b \approx (a\ b) \cdot (a\ b) \cdot a \approx a$$

  gives the dual form of (S_3).

$$\frac{}{\Sigma : \Gamma, A \Rightarrow A, \Delta}\ hyp$$

$$\frac{}{\Sigma : \Gamma \Rightarrow \top, \Delta}\ \top R \qquad \frac{}{\Sigma : \Gamma, \bot \Rightarrow \Delta}\ \bot L$$

$$\frac{\Sigma : \Gamma \Rightarrow \phi, \Delta \quad \Sigma : \Gamma \Rightarrow \psi, \Delta}{\Sigma : \Gamma \Rightarrow \phi \wedge \psi, \Delta}\ \wedge R \qquad \frac{\Sigma : \Gamma, \phi_1, \phi_2 \Rightarrow \Delta}{\Sigma : \Gamma, \phi_1 \wedge \phi_2 \Rightarrow \Delta}\ \wedge L$$

$$\frac{\Sigma : \Gamma \Rightarrow \phi_1, \phi_2, \Delta}{\Sigma : \Gamma \Rightarrow \phi_1 \vee \phi_2, \Delta}\ \vee R \qquad \frac{\Sigma : \Gamma, \phi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Sigma : \Gamma, \phi \vee \psi \Rightarrow \Delta}\ \vee L$$

$$\frac{\Sigma : \Gamma, \phi \Rightarrow \psi, \Delta}{\Sigma : \Gamma \Rightarrow \phi \supset \psi, \Delta}\ \supset R \qquad \frac{\Sigma : \Gamma \Rightarrow \phi, \Delta \quad \Sigma : \Gamma, \psi \Rightarrow \Delta}{\Sigma : \Gamma, \phi \supset \psi \Rightarrow \Delta}\ \supset L$$

$$\frac{\Sigma : \Gamma, \phi \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \neg\phi, \Delta}\ \neg R \qquad \frac{\Sigma : \Gamma \Rightarrow \phi, \Delta}{\Sigma : \Gamma, \neg\phi \Rightarrow \Delta}\ \neg L$$

$$\frac{\Sigma, x : \Gamma \Rightarrow \phi, \Delta \quad (x \notin \Sigma)}{\Sigma : \Gamma \Rightarrow \forall x.\phi, \Delta}\ \forall R \qquad \frac{\Sigma \vdash t : \sigma \quad \Sigma : \Gamma, \forall x^\sigma.\phi, \phi\{t/x\} \Rightarrow \Delta}{\Sigma : \Gamma, \forall x^\sigma.\phi \Rightarrow \Delta}\ \forall L$$

$$\frac{\Sigma \vdash t : \sigma \quad \Sigma : \Gamma \Rightarrow \exists x^\sigma.\phi, \phi\{t/x\}, \Delta}{\Sigma : \Gamma \Rightarrow \exists x^\sigma.\phi, \Delta}\ \exists R \qquad \frac{\Sigma, x : \Gamma, \phi \Rightarrow \Delta \quad (x \notin \Sigma)}{\Sigma : \Gamma, \exists x.\phi \Rightarrow \Delta}\ \exists L$$

$$\frac{\Sigma : \Gamma, t \approx t \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \Delta}\ \approx R \qquad \frac{\Sigma : \Gamma, t \approx u, \phi(t), \phi(u) \Rightarrow \Delta}{\Sigma : \Gamma, t \approx u, \phi(t) \Rightarrow \Delta}\ \approx S$$

Figure 4.4: Sequent calculus for classical first-order equational logic

$$\frac{}{\Sigma : \Gamma, \phi \Rightarrow \phi, \Delta}\ hyp^* \qquad \frac{\Sigma : \Gamma \Rightarrow \phi, \Delta \quad \Sigma : \Gamma', \phi \Rightarrow \Delta'}{\Sigma : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'}\ cut$$

$$\frac{\Sigma : \Gamma, (a\ b) \cdot \phi \Rightarrow \Delta}{\Sigma : \Gamma, \phi \Rightarrow \Delta}\ EVL \qquad \frac{\Sigma : \Gamma \Rightarrow (a\ b) \cdot \phi, \Delta}{\Sigma : \Gamma \Rightarrow \Delta, \phi}\ EVR$$

Figure 4.5: Some admissible rules of $\mathrm{NL}^{\Rightarrow}$

(S_1)	$(a\ a) \cdot x \approx x$
(S_2)	$(a\ b) \cdot (a\ b) \cdot x \approx x$
(S_3)	$(a\ b) \cdot a \approx a$
$(E_{\langle\rangle})$	$(a\ b) \cdot \langle\rangle \approx \langle\rangle$
$(E_\times)$	$(a\ b) \cdot \langle x, y\rangle \approx \langle (a\ b) \cdot x, (a\ b) \cdot y\rangle$
(E_f)	$(a\ b) \cdot f(x) \approx f((a\ b) \cdot x)$
(E_p)	$p(x) \supset p((a\ b) \cdot x)$
(F_1)	$a \# x \wedge b \# x \supset (a\ b) \cdot x \approx x$
(F_2)	$a^\nu \# b^{\nu'} \quad (\nu \not\equiv \nu')$
(F_{3a})	$a \# a \supset \bot$
(F_{3b})	$a \# b \vee a \approx b$
(A_1)	$a \# x \wedge x \approx (a\ b) \cdot y \supset \langle a\rangle x \approx \langle b\rangle y$
(U)	$x^{\mathbf{1}} \approx \langle\rangle$

Figure 4.6: Equational and freshness axioms

$$\frac{\Sigma : \Gamma, P, Q_1 \Rightarrow \Delta \quad \cdots \quad \Sigma : \Gamma, P, Q_n \Rightarrow \Delta}{\Sigma : \Gamma, P \Rightarrow \Delta}\ Ax \qquad P \supset \bigvee Q \text{ an axiom instance}$$

$$\frac{\Sigma \vdash t : \langle\nu\rangle\sigma \quad \Sigma\#\mathsf{a}^\nu, x^\sigma : \Gamma, t \approx \langle\mathsf{a}\rangle x \Rightarrow \Delta \quad (\mathsf{a}, x \notin \Sigma)}{\Sigma : \Gamma \Rightarrow \Delta}\ A_2$$

$$\frac{\Sigma \vdash t : \sigma_1 \times \sigma_2 \quad \Sigma, x_1^{\sigma_1}, x_2^{\sigma_2} : \Gamma, t \approx \langle x_1, x_2\rangle \Rightarrow \Delta \quad (x_1, x_2 \notin \Sigma)}{\Sigma : \Gamma \Rightarrow \Delta}\ P$$

$$\frac{\Sigma|_{\mathsf{a}} \vdash u : \sigma \quad \Sigma : \Gamma, \mathsf{a} \# u \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \Delta}\ \Sigma\# \qquad \frac{\Sigma\#\mathsf{a} : \Gamma \Rightarrow \Delta \quad (\mathsf{a} \notin \Sigma)}{\Sigma : \Gamma \Rightarrow \Delta}\ F$$

$$\frac{\Sigma\#\mathsf{a} : \Gamma \Rightarrow \phi, \Delta \quad (\mathsf{a} \notin \Sigma)}{\Sigma : \Gamma \Rightarrow \text{И}\mathsf{a}.\phi, \Delta}\ \text{И}R \qquad \frac{\Sigma\#\mathsf{a} : \Gamma, \phi \Rightarrow \Delta \quad (\mathsf{a} \notin \Sigma)}{\Sigma : \Gamma, \text{И}\mathsf{a}.\phi \Rightarrow \Delta}\ \text{И}L$$

Figure 4.7: $\mathrm{NL}^{\Rightarrow}$ sequent rules

- $(E_{\langle\rangle})$, $(E_\times)$, (E_f), (E_p): These *equivariance* axioms assert that swapping has no effect on constants, commutes with built-in function symbols, and preserves the validity of atomic formulas. Together these amount to the **Equivariance principle** mentioned in Chapter 1. Note that (E_f) includes equivariance for swapping and abstraction and (E_p) includes equivariance for freshness and equality as special cases.

- (F_1)–(F_3): These *freshness* axioms define the behavior of the $\#$ predicate and relate it to swapping. (F_1) asserts that swapping two fresh names has no effect; (F_2) that different name-sorts are disjoint; and $(F_{3a}$–$F_{3b})$ that for pairs of names, freshness coincides with inequality.

- $(\Sigma\#)$: This rule can be used to extract freshness information from the context Σ. Intuitively, $\Sigma|_{\mathsf{a}}$ is Σ with all variables that might mention a removed. The $\Sigma\#$ rule and $\Sigma|_{\mathsf{a}}$ notation are discussed further in Section 4.3.4.

- (F): Read bottom-up this rule says we can always generate a completely fresh name a. This embodies the **Freshness principle** mentioned in Chapter 1.

- (A_1): This axiom indicates that abstractions are equal up to α-equivalence. Two abstractions are equal provided their bodies are equal up to swapping the abstracted names and the name abstracted on one side is fresh for the other. This defines α-equivalence in terms of swapping. This is a logical version of Definition 3.3.18.

- (U), (P), (A_2) These rules are surjectivity laws for units, pairing, and abstraction. The (U) rule says that the only element of the unit sort is $\langle\rangle$. The (P) rule says that any element of the pair sort can be decomposed into a pair. The (A_2) rule is a surjectivity property indicating that any abstraction can be "freshened" to a fresh name-constant a and some body x.

- $(\approx R), (\approx S)$: These rules deal with equality. The $(\approx R)$ rule indicates that equality is reflexive, whereas $(\approx S)$ is the substitution property. The latter can be used to derive symmetry, transitivity and congruence properties of equality as well [93].

Note that the numbering of the A, F, and S axiom groups reflects that of the axioms of Pitts' system, for comparison with [108].

4.3.3 The New-Quantifier Rules

The rules $\reflectbox{N}R$ and $\reflectbox{N}L$ are symmetric; we consider just $\reflectbox{N}R$. The $\reflectbox{N}$-quantifier ought to satisfy the "some/any" equivalences

$$\reflectbox{N}\mathsf{a}.\phi(\mathsf{a},\vec{x}) \Leftrightarrow \exists a.a \# \vec{x} \wedge \phi(a,\vec{x}) \Leftrightarrow \forall a.a \# \vec{x} \supset \phi(a,\vec{x}) \tag{4.14}$$

where $\vec{x} = FV(\text{И}\mathsf{a}.\phi)$. So at first sight a sequent rule such as

$$\frac{\Gamma, \mathsf{a} \# \vec{x} \Rightarrow \phi}{\Gamma \Rightarrow \text{И}\mathsf{a}.\phi} \tag{4.15}$$

(where $FV(\text{И}\mathsf{a}.\phi) = \vec{x}$ and a is fresh) might seem appropriate. Such rules were proposed by Gabbay and Pitts [42, 108] in earlier papers on FM and nominal logic. However, these rules are not well-behaved with respect to substitution. Deduction rules should be closed under substitution; this property greatly simplifies the proof of the cut-elimination cases for $\forall$ and $\exists$. But (4.15) is not closed under substitution, for substituting a non-variable term such as a constant c for one of the $\vec{x}$ results in a non-instance. The obvious approach to introducing a rule for the Freshness Principle has the same problem.

One approach to fixing this problem (due to Gabbay, and used in [40, 41]) is to generalize the rule to something like

$$\frac{\Gamma, \mathsf{a} \# \vec{t} \Rightarrow \phi(\mathsf{a}, \vec{t})}{\Gamma \Rightarrow \text{И}\mathsf{a}.\phi(\mathsf{a}, \vec{t})} \tag{4.16}$$

where $\phi(\mathsf{a}, \vec{t})$ is a decomposition of ϕ into a name-free context with $n+1$ holes, one for a and one for each of the n terms $\vec{t}$. This decomposition is called a *slice* of ϕ over a, and has the desirable property that if $\phi(\mathsf{a}, \vec{t})$ is a slice, then $\phi(\mathsf{a}, \vec{t})\{u/x\} = \phi(\mathsf{a}, \vec{t}\{u/x\})$ is also a slice. This results in a rule that is closed under substitution, but the resulting system is rather difficult to explain and analyze.

We propose a much cleaner approach using a structured context Σ to summarize all the information about freshness that is needed for the freshness rule F and И-rules. To deal with these rules, all we need to know is the order in which variables and name-constants have been introduced. A name-constant that has been introduced by И more recently than a variable can be assumed to be fresh for that variable. Our rule for ИR is therefore

$$\frac{\Sigma \# \mathsf{a} : \Gamma \Rightarrow \phi}{\Sigma : \Gamma \Rightarrow \text{И}\mathsf{a}.\phi}\ \text{И}R \tag{4.17}$$

Intuitively, this rule states that to prove И$\mathsf{a}.\phi$ from Γ, it suffices to show ϕ from Γ under the assumption that a is completely fresh in the current context. On the other hand the left-rule

$$\frac{\Sigma \# \mathsf{a} : \Gamma, \phi \Rightarrow \psi}{\Sigma : \Gamma, \text{И}\mathsf{a}.\phi \Rightarrow \psi}\ \text{И}L \tag{4.18}$$

says that to prove ψ from Γ and И$\mathsf{a}.\phi$, it suffices to prove ψ from Γ and ϕ assuming that a is completely fresh. (Compare these rules with the informal proof principles for И discussed in Section 3.2.5). In both cases, recall that И-quantified formulas are subject to α-equivalence, so there is no loss of generality involved in assuming that the bound variable is fresh (does not appear in Σ).

Both the rules and the cut-elimination proof of NL$^{\Rightarrow}$ are more straightforward than for any other proof system for nominal logic. The price that we pay for this simplicity is that the semantics is more complicated because of the presence of an explicit context. However, this is essentially a cosmetic difference, and keeping track of variables in the semantic judgments has advantages as well.

4.3.4 Contexts and Freshness

We formally define the notation $\Sigma|_{\mathsf{a}}$ used in the $\Sigma\#$ rule.

Definition 4.3.5 (Context restriction). *Let Σ be a context and $\mathsf{a} \in \Sigma$. Then the context restriction of Σ to a is defined recursively as*

$$\begin{aligned} \Sigma\#\mathsf{a}|_{\mathsf{a}} &= \Sigma \\ \Sigma\#\mathsf{b}|_{\mathsf{a}} &= \Sigma|_{\mathsf{a}}\#\mathsf{b} \\ \Sigma, x|_{\mathsf{a}} &= \Sigma|_{\mathsf{a}} \end{aligned}$$

In words, the context restriction removes all variables from Σ that may mention a, that is, those to the right of a. The $\Sigma\#$ rule says that it is safe to assume $\mathsf{a} \# t$ if t is well-formed in $\Sigma|_{\mathsf{a}}$. This allows us to use freshness information derivable from Σ in a proof. Note that name-constants on either side of a are guaranteed to be different from a, so the restriction operation only needs to remove variables.

Contexts are equivalent up to rearranging variables and name-constants as long as freshness relationships do not change. For example, $x, y\#\mathsf{a}\#\mathsf{b}$ and $y, x\#\mathsf{b}\#\mathsf{a}$ are logically equivalent contexts. In addition, some contexts are "stronger" (more restrictive) than others, for example $x, y\#\mathsf{a}\#\mathsf{b}$ is more restrictive than $x\#\mathsf{a}, y\#\mathsf{b}$, since a may appear in y in the latter but not the former. We now make these intuitive notions formal.

Definition 4.3.6. *We define the* is weaker than *relation $\leq$ on contexts as the least partial order satisfying:*

1. $\Sigma, x, y \leq \Sigma, y, x$
2. $\Sigma\#\mathsf{a}\#\mathsf{b} \leq \Sigma\#\mathsf{b}\#\mathsf{a}$
3. $\Sigma\#\mathsf{a}, x \leq \Sigma, x\#\mathsf{a}$
4. $\Sigma \leq \Sigma\#\mathsf{a}$ *(if $\mathsf{a} \notin \Sigma$)*
5. $\Sigma \leq \Sigma, x$ *(if $x \notin \Sigma$)*
6. $\Sigma \leq \Sigma'$ *implies* $\Sigma; \Sigma'' \leq \Sigma'; \Sigma''$

We say Σ, Σ' are equivalent *(written $\Sigma \equiv \Sigma'$) if $\Sigma \leq \Sigma'$ and $\Sigma' \leq \Sigma$.*

Clearly, $\equiv$ is a congruence respecting the structure of contexts, by property (6). Moreover, $\Sigma \leq \Sigma'$ if and only if there is a finite chain of uses of reflexivity, transitivity, and properties (1)–(6) above.

Context restriction respects weakening.

Lemma 4.3.7. *If* $\mathsf{a} \in \Sigma \leq \Sigma'$ *then* $\Sigma|_{\mathsf{a}} \leq \Sigma'|_{\mathsf{a}}$.

Proof. The proof is by induction on the derivation of $\Sigma \leq \Sigma'$. The only interesting cases are the ones involving a.

In the case of $\Sigma\#\mathsf{a}\#\mathsf{b} \leq \Sigma\#\mathsf{b}\#\mathsf{a}$, we have $\Sigma\#\mathsf{a}\#\mathsf{b}|_{\mathsf{a}} = \Sigma\#\mathsf{b} = \Sigma\#\mathsf{b}\#\mathsf{a}|_{\mathsf{a}}$.

If $\Sigma\#\mathsf{a}, x \leq \Sigma, x\#\mathsf{a}$ then $\Sigma\#\mathsf{a}, x|_{\mathsf{a}} = \Sigma \leq \Sigma, x = \Sigma, x\#\mathsf{a}|_{\mathsf{a}}$.

If $\Sigma \leq \Sigma\#\mathsf{b}$ then $\mathsf{b} \neq \mathsf{a}$ so $\Sigma|_{\mathsf{a}} \leq \Sigma|_{\mathsf{a}}\#\mathsf{b} = \Sigma\#\mathsf{b}|_{\mathsf{a}}$. □

Proposition 4.3.8 (Context weakening). *If* $\Sigma \leq \Sigma'$ *and*

1. $\Sigma \vdash t : \sigma$ *then* $\Sigma' \vdash t : \sigma$
2. $\Sigma \vdash \phi : \mathbf{prop}$ *then* $\Sigma' \vdash \phi : \mathbf{prop}$
3. $\Sigma : \Gamma \Rightarrow \Delta$ *then* $\Sigma' : \Gamma \Rightarrow \Delta$.

Proof. Parts (1) and (2) are trivial since the order of the names and variables in Σ is irrelevant to typing.

Part (3) is a straightforward induction on the derivation, using parts (1), (2), and Lemma 4.3.7 as appropriate. □

From now on we may use this property whenever convenient without explicit citation.

4.3.5 Example Derivations

We first show an important property of И: namely, it is equivalent to either an existential or universal formula.

Lemma 4.3.9. *Let* $\phi(b)$ *be a well-formed formula in* Σ, b^ν. *Let* $b \# \Sigma$ *abbreviate* $b \# \vec{x} \wedge b \# \vec{\mathsf{b}}$, *where* $\vec{x} = FV(\Sigma)$ *and* $\vec{\mathsf{b}} = FN(\Sigma)$. *Assume* $\mathsf{a}^\nu \notin \Sigma$. *Then the following are equivalent:*

$$\forall b. b \# \Sigma \supset \phi(b) \tag{4.19}$$

$$\text{И}\mathsf{a}.\phi(\mathsf{a}) \tag{4.20}$$

$$\exists b. b \# \Sigma \wedge \phi(b) \tag{4.21}$$

Proof. First, note that $\Sigma\#\mathsf{a} \vdash \mathsf{a} \# \Sigma$.

To prove (4.20) from (4.19), we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Sigma\#\mathsf{a} : \cdot \Rightarrow \mathsf{a} \# \Sigma}\ \Sigma\#, hyp \qquad \dfrac{}{\Sigma\#\mathsf{a} : \phi(\mathsf{a}) \Rightarrow \phi(\mathsf{a})}\ hyp}{\Sigma\#\mathsf{a} : \mathsf{a} \# \Sigma \supset \phi(\mathsf{a}) \Rightarrow \phi(\mathsf{a})}\ {\supset}L}{\Sigma\#\mathsf{a} : \forall b. b \# \Sigma \supset \phi(b) \Rightarrow \phi(\mathsf{a})}\ \forall L}{\Sigma : \forall b. b \# \Sigma \supset \phi(b) \Rightarrow \text{И}\mathsf{a}.\phi(\mathsf{a})}\ \text{И}R$$

To prove (4.21) from (4.20), we have the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Sigma\#\mathsf{a} : \cdot \Rightarrow \mathsf{a} \# \Sigma}\ \Sigma\#, hyp \qquad \dfrac{}{\Sigma\#\mathsf{a} : \phi(\mathsf{a}) \Rightarrow \phi(\mathsf{a})}\ hyp}{\Sigma\#\mathsf{a} : \phi(\mathsf{a}) \Rightarrow \mathsf{a} \# \Sigma \wedge \phi(\mathsf{a})}\ \wedge R}{\Sigma\#\mathsf{a} : \phi(\mathsf{a}) \Rightarrow \exists b.b \# \Sigma \wedge \phi(b)}\ \exists R}{\Sigma : \text{И}\mathsf{a}.\phi(\mathsf{a}) \Rightarrow \exists b.b \# \Sigma \wedge \phi(b)}\ \text{И}L$$

To prove (4.19) from (4.21), we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma, b, b' : \phi(b') \Rightarrow \phi(b')}\ hyp}{\Sigma, b, b' : b' \# \Sigma, b \# \Sigma, (b\ b') \cdot \phi(b) \Rightarrow \phi(b')}\ \approx}{\Sigma, b, b' : b' \# \Sigma, b \# \Sigma, \phi(b) \Rightarrow \phi(b')}\ EV}{\Sigma, b, x : b \# \Sigma, \phi(b) \Rightarrow b' \# \Sigma \supset \phi(b')}\ \supset R}{\Sigma, b : b \# \Sigma, \phi(b) \Rightarrow \forall b.b \# \Sigma \supset \phi(b)}\ \forall R}{\Sigma, b : b \# \Sigma \wedge \phi(b) \Rightarrow \forall b.b \# \Sigma \supset \phi(b)}\ \wedge L}{\Sigma : \exists b.b \# \Sigma \wedge \phi(b) \Rightarrow \forall b.b \# \Sigma \supset \phi(b)}\ \exists L$$

This completes the proof. □

We now provide several additional derivations of valid sequents in NL.

1. $\Sigma : \cdot \Rightarrow \text{И}\mathsf{a}.\mathsf{a} \# x$, where $x \in \Sigma$.

$$\dfrac{\dfrac{\dfrac{}{\Sigma\#\mathsf{a} : \mathsf{a} \# x \cdot \Rightarrow \mathsf{a} \# x}}{\Sigma\#\mathsf{a} : \cdot \Rightarrow \mathsf{a} \# x}}{\Sigma : \cdot \Rightarrow \text{И}\mathsf{a}.\mathsf{a} \# x}\ \text{И}R$$

2. $\Sigma : \cdot \Rightarrow \text{И}\mathsf{a}.\text{И}\mathsf{b}.(\mathsf{a}\ \mathsf{b}) \cdot x \approx x$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma\#\mathsf{a}\#\mathsf{b} : (\mathsf{a}\ \mathsf{b}) \cdot x \approx x \Rightarrow (\mathsf{a}\ \mathsf{b}) \cdot x \approx x}\ Ax}{\Sigma\#\mathsf{a}\#\mathsf{b} : \cdot \Rightarrow (\mathsf{a}\ \mathsf{b}) \cdot x \approx x}\ \text{И}R}{\Sigma\#\mathsf{a} : \cdot \Rightarrow \text{И}\mathsf{b}.(\mathsf{a}\ \mathsf{b}) \cdot x \approx x}\ \text{И}R}{\Sigma : \cdot \Rightarrow \text{И}\mathsf{a}.\text{И}\mathsf{b}.(\mathsf{a}\ \mathsf{b}) \cdot x \approx x}$$

3. $\Sigma : a \# x \Rightarrow \text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot x \approx x$

$$\dfrac{\dfrac{\dfrac{}{\Sigma\#\mathsf{b} : a \# x, (a\ \mathsf{b}) \cdot x \approx x \Rightarrow (a\ \mathsf{b}) \cdot x \approx x}\ Ax}{\Sigma\#\mathsf{b} : a \# x \Rightarrow (a\ \mathsf{b}) \cdot x \approx x}\ \text{И}R}{\Sigma : a \# x \Rightarrow \text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot x \approx x}$$

4. $\Sigma : \text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot x \approx x \Rightarrow a \# x$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma\#\mathsf{b} : a \# x \Rightarrow a \# x}\ EV, Ax}{\Sigma\#\mathsf{b} : \mathsf{b} \# (a\ \mathsf{b}) \cdot x \Rightarrow a \# x}\ \approx S}{\Sigma\#\mathsf{b} : (a\ \mathsf{b}) \cdot x \approx x \Rightarrow a \# x}\ \text{И}L}{\Sigma : \text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot x \approx x \Rightarrow a \# x}$$

4.4 Proof-Theoretic Properties

In this section we will discuss some of the proof-theoretic properties of NL$^{\Rightarrow}$. First, we show that several structural rules are admissible in NL. We have already seen several examples, namely contraction, weakening, substitution, and renaming lemmas. The proofs of these properties are straightforward extensions of the proofs for ordinary first-order logic. In addition, we show that the general hypothesis and equivariance rules (in which the principal formula is arbitrary) are admissible assuming only atomic versions of these rules. Next, we show cut-elimination. We conclude with a discussion of issues such as consistency, Skolemization, and the relationship of NL$^{\Rightarrow}$ to Pitts' axiomatization.

4.4.1 General Equivariance and Hypothesis Rules

The equivariance axiom shown in Figure 4.6 and the hypothesis rule in Figure 4.4 are apparently restrictive in that the hypothesis rule and the equivariance axiom are restricted to apply only to atomic formulas. This restriction greatly simplifies proof search, as well as simplifying the proof of cut-elimination given in the next section. Nevertheless, it is not a real restriction, since general equivariance rules (on the right and left) and a general hypothesis rule are admissible.

Lemma 4.4.1 (Admissibility of EVL, EVR). *The EVL and EVR rules*

$$\frac{\Sigma : \Gamma, (a\ b) \cdot \phi \Rightarrow \Delta}{\Sigma : \Gamma, \phi \Rightarrow \Delta}\ EVL \qquad \frac{\Sigma : \Gamma \Rightarrow (a\ b) \cdot \phi, \Delta}{\Sigma : \Gamma \Rightarrow \phi, \Delta}\ EVR$$

where ϕ is an arbitrary formula, are admissible.

Proof. The proof is by induction on the *weight* of a derivation: the number of logical rules (that is, *hyp* and left and right rules) used in the derivation. We proceed by induction to show that if the hypothesis of an instance of EVL or EVR has a derivation then the conclusion of the respective rule has a derivation of the same weight.

We first consider EVL. The only interesting cases are when $(a\ b) \cdot \phi$ is principal on the left, otherwise the induction step is straightforward. Furthermore, only the cases for *hyp* and $\supset L$ are nontrivial.

If the derivation is of the form

$$\frac{}{\Gamma, (a\ b) \cdot A \Rightarrow (a\ b) \cdot A, \Delta}$$

then we may derive $\Gamma, A \Rightarrow (a\ b) \cdot A, \Delta$ as follows:

$$\frac{\Sigma : \Gamma, (a\ b) \cdot A \Rightarrow (a\ b) \cdot A, \Delta}{\Sigma : \Gamma, A \Rightarrow (a\ b) \cdot A, \Delta}\ E_p$$

This derivation has the same weight, 1, as the first.

If the derivation is of the form

$$\dfrac{\Sigma : \Gamma, (a\ b) \cdot P \supset (a\ b) \cdot Q \Rightarrow (a\ b) \cdot P, \Delta \quad \Sigma : \Gamma, (a\ b) \cdot Q \Rightarrow \Delta}{\Sigma : \Gamma, (a\ b) \cdot P \supset (a\ b) \cdot Q \Rightarrow \Delta} \supset L$$

then using the admissibility of EVR and EVL on the left and EVR on the right we obtain

$$\dfrac{\dfrac{\Sigma : \Gamma, (a\ b) \cdot P \supset (a\ b) \cdot Q \Rightarrow (a\ b) \cdot P, \Delta}{\Sigma : \Gamma, P \supset Q \Rightarrow P, \Delta} EVL, EVR \quad \dfrac{\Sigma : \Gamma, (a\ b) \cdot Q \Rightarrow \Delta}{\Sigma : \Gamma, Q \Rightarrow \Delta} EVL}{\Sigma : \Gamma, P \supset Q \Rightarrow \Delta} \supset L$$

This transformation is obviously weight-preserving by induction.

For EVR, the interesting cases are those for *hyp* and $\supset R$ where $(a\ b) \cdot \phi$ is principal on the right. Suppose the derivation is of the form

$$\dfrac{}{\Gamma, (a\ b) \cdot A \Rightarrow (a\ b) \cdot A, \Delta}$$

Then we can derive

$$\dfrac{\dfrac{}{\Gamma, (a\ b) \cdot (a\ b) \cdot A \Rightarrow A, \Delta} \approx, hyp}{\Gamma, (a\ b) \cdot A \Rightarrow A, \Delta} E_p$$

This derivation has the same weight, 1, as the first.

If the derivation is of the form

$$\dfrac{\Gamma, (a\ b) \cdot P \Rightarrow (a\ b) \cdot Q, \Delta}{\Gamma \Rightarrow (a\ b) \cdot P \supset (a\ b) \cdot Q, \Delta} \supset R$$

then since EVL and EVR are admissible for all subderivations of this derivation, by induction we can derive

$$\dfrac{\dfrac{\Gamma, (a\ b) \cdot P \Rightarrow (a\ b) \cdot Q, \Delta}{\Gamma, P \Rightarrow Q, \Delta} EVL, EVR}{\Gamma \Rightarrow P \supset Q, \Delta} \supset R$$

This transformation is obviously weight-preserving by induction. □

Lemma 4.4.2 (Admissibility of *hyp).** *The hyp* rule*

$$\dfrac{}{\Sigma : \Gamma, \phi \Rightarrow \phi, \Delta} hyp^*$$

where ϕ is an arbitrary formula, is admissible.

Proof. The cases for the ordinary connectives of first-order logic are standard. The case for $\phi = \text{И}\mathsf{a}.P$ is as follows. By induction, we may assume that $\Sigma\#\mathsf{a}\#\mathsf{a}' : \Gamma, P(\mathsf{a}') \Rightarrow P(\mathsf{a}'), \Delta$ is derivable. We derive

$$\dfrac{\dfrac{\dfrac{\Sigma\#\mathsf{a}\#\mathsf{a}' : \Gamma, P(\mathsf{a}') \Rightarrow P(\mathsf{a}'), \Delta}{\Sigma\#\mathsf{a}\#\mathsf{a}' : \Gamma, P(\mathsf{a}) \Rightarrow P(\mathsf{a}'), \Delta} EVL}{\Sigma\#\mathsf{a} : \Gamma, P(\mathsf{a}) \Rightarrow \text{И}\mathsf{a}.P, \Delta} \text{И}R}{\Sigma : \Gamma, \text{И}\mathsf{a}.P \Rightarrow \text{И}\mathsf{a}.P, \Delta} \text{И}L$$

where nonlogical axioms are used to derive $(\mathsf{a}\ \mathsf{a}') \cdot P(\mathsf{a}) = P(\mathsf{a}')$ from $P(\mathsf{a})$. Using the induction hypothesis, the judgment $\Sigma\#\mathsf{a}\#\mathsf{a}' : \Gamma, P(\mathsf{a}') \Rightarrow P(\mathsf{a}'), \Delta$ is derivable, since it is an instance of *hyp** with a smaller principal formula. □

4.4.2 Cut-Elimination

In this section we show that the cut-rule can be eliminated from derivations. This is an important property that ensures that NL is a sensible sequent proof system.

Lemma 4.4.3 (Admissibility of cut). *If* $\Sigma : \Gamma \Rightarrow \Delta, \phi$ *and* $\Sigma : \Gamma', \phi \Rightarrow \Delta'$ *have cut-free derivations then so does* $\Sigma : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'$.

Proof. As usual, the proof is by induction on the complexity of the cut formula ϕ, and a sub-induction on the sum of the heights of the derivations Π of $\Gamma \Rightarrow \Delta, \phi$ and Π' of $\Gamma', \phi \Rightarrow \Delta'$. That is, in each case, we may assume the induction hypothesis holds for any less complex cut-formula ϕ, or for the same cut-formula with smaller derivations in place of Π or Π'.

The cases are divided into several (nonexclusive) classes:

- Base cases in which one of the derivations is an axiom;
- Right-commuting cases in which the cut formula is not principal in Π;
- Left-commuting cases in which the cut formula is not principal in Π',
- Principal cases in which the cut formula is principal on both sides.

Many of the cases, however, involve only standard rules of first-order equational logic, and are completely standard (see [93] for detailed proofs of these cases for a similar sequent calculus). The new left-commuting and right-commuting cases are straightforward. We give an example of a new case:

If Π' ends with $\mathsf{И}L$ and the cut-formula is not principal, i.e. we have

$$\frac{\Sigma\#\mathsf{a} : \Gamma, \phi, \psi \Rightarrow \Delta}{\Sigma : \Gamma, \phi, \mathsf{И}\mathsf{a}.\psi \Rightarrow \Delta}\ \mathsf{И}L$$

then we may derive

$$\cfrac{\cfrac{\Sigma\#\mathsf{a} : \Gamma, \psi \Rightarrow \Delta, \phi \quad \Sigma\#\mathsf{a} : \Gamma, \phi, \psi \Rightarrow \Delta}{\Sigma\#\mathsf{a} : \Gamma, \psi \Rightarrow \Delta}\ \mathit{cut}}{\Sigma : \Gamma, \mathsf{И}\mathsf{a}.\psi \Rightarrow \Delta}\ \mathsf{И}L$$

where we obtain $\Sigma\#\mathsf{a} : \Gamma, \psi \Rightarrow \Delta$ from $\Sigma : \Gamma, \mathsf{И}\mathsf{a}.\psi \Rightarrow \Delta$ using an easily derived inversion principle for $\mathsf{И}L$.

This leaves the possibility of new principal cut cases; in fact, there is only one new principal cut, arising from $\mathsf{И}R$ and $\mathsf{И}L$.

In this case, the derivations are of the form

$$\dfrac{\begin{array}{c}\Pi\\ \Sigma\#\mathsf{a} : \Gamma \Rightarrow \phi, \Delta\end{array}}{\Sigma : \Gamma \Rightarrow \Delta, \text{И}\mathsf{a}.\phi}\,\text{И}R \qquad \dfrac{\begin{array}{c}\Pi'\\ \Sigma\#\mathsf{a} : \Gamma', \phi \Rightarrow \Delta'\end{array}}{\Sigma : \Gamma', \text{И}\mathsf{a}.\phi \Rightarrow \Delta'}\,\text{И}L$$

where without loss of generality we assume that the same fresh name $\mathsf{a} \notin \Sigma$ was used for in both sub-derivations. Since the formula ϕ is smaller than $\text{И}\mathsf{a}.\phi$, we can obtain a derivation Π'' of $\Sigma\#\mathsf{a} : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'$ from Π and Π' by the induction hypothesis. Then using rule F we may derive

$$\dfrac{\begin{array}{c}\Pi''\\ \Sigma\#\mathsf{a} : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'\end{array}}{\Sigma : \Gamma, \Gamma' \Rightarrow \Delta, \Delta'}\,F$$

This completes the proof for this case. □

Theorem 4.4.4 (Cut-elimination). *If $\Gamma \Rightarrow \Delta$ has any derivation then it has a cut-free derivation.*

Proof. The proof is by induction on the number of cuts in a derivation. If there are none then the derivation is already cut-free. If any derivation using n cuts has a cut-free derivation, suppose we have a derivation Π using $n+1$ cuts. There must be a cut rule whose subderivations are cut-free, that is, $\Pi = \Pi[\Pi']$ where Π' starts with a cut and is cut-free. Using Lemma 4.4.3, we can obtain a cut-free derivation Π'' of the conclusion of this cut. Replacing this derivation in Π, we have $\Pi[\Pi'']$, a derivation using n cuts, to which the induction hypothesis applies. □

4.4.3 Other Issues

Syntactic Consistency

For many logics, *consistency* is an important immediate consequence of cut-elimination, since there is obviously no cut-free proof of the sequent $\Rightarrow \bot$. This method of proving consistency is especially useful for logics motivated in terms of proof theory, such as linear logic and intuitionistic logic, since the proof theories of such logics are much simpler than their model theories.

Syntactic consistency is less obvious for a sequent calculus with nonlogical rules, such as $\mathrm{NL}^{\Rightarrow}$, since the nonlogical rules might be inconsistent. However, Negri and von Plato have shown that for a sequent calculus with non-logical rules, if there is a derivation of $\Gamma \Rightarrow \bot$ then there is such a derivation using nonlogical rules alone [93, Ch. 6]. This reduces consistency of the logic to consistency of the nonlogical axioms. The same technique can be used to show that $\mathrm{NL}^{\Rightarrow}$ is consistent, since the nonlogical rules of $\mathrm{NL}^{\Rightarrow}$ are valid in term models of nominal logic. This fact is a special case of Herbrand's theorem for nominal logic which will be proved in the next chapter.

Skolemization

In first-order classical logic, it is safe to remove existential quantifiers from formulas by introducing new function symbols. For example, we can replace a formula such as $\forall x.\exists y.\forall z.P(x,y,z)$ with $\forall x.\forall z.P(x,f(x),z)$ where f is a new function symbol. This can be viewed as a *choice principle* for first-order logic analogous to the Axiom of Choice in set theory. In higher-order logic, this choice principle can be given as an explicit theorem, using a higher-order function f:

$$\forall x^a.\exists y^b.P(x,y) \supset \exists f^{a\to b}.\forall x^a.P(x,f(x)) \ . \tag{4.22}$$

Since nominal logic is based on ideas from FM-set theory, in which the Axiom of Choice may not hold, it should not be surprising that Skolemization is not valid in NL. However, the reason is rather technical. Suppose a theory Γ contains an axiom $\phi = \forall x.\exists a.a \# x$. If we add a function symbol f to a nominal language and replace the above axiom with $\phi' = \forall x.f(x) \# x$ then we get an inconsistent theory. The reason is that while a function f mapping each x to a name fresh for x exists (using the Axiom of Choice), such a function cannot be equivariant, but all the function symbols of a nominal language are assumed to be equivariant. Moreover, a fresh name choice function on a name set A, $f : A \to A$, cannot even be $\mathcal{I}$-supported, since an $\mathcal{I}$-supported function on A must be constant everywhere except a small subset of $\mathcal{I}$, whereas a fresh name choice function cannot be constant anywhere. So the higher-order choice principle would not be valid in a higher-order nominal logic either.

On the other hand, as Pitts has observed, there is no problem with unique choice principles in NL. In addition, there seems to be no problem with choice principles over "closed" sorts, that is, when the objects being chosen are guaranteed to have empty support.

Relationship with Pitts' Nominal Logic

It is a natural question to ask whether the sequent calculus $\mathrm{NL}^{\Rightarrow}$ really defines the same logic as Pitts' axiomatization (up to minor differences such as the use of name-constants in И rather than variables). We believe that these minor differences can be easily reconciled, and Pitts' system shown to be equivalent to ours.

4.5 Notes

The approach to dealing with nonlogical axioms within a sequent calculus employed in this chapter is due to Negri and von Plato (see [93, Ch. 6]).

Pitts' original version of nominal logic was the basis of this work. Our version of NL was also influenced by other systems, including the *nominal equational logic* used in nominal unification and rewriting [36, 126, 127]. The sequent calculus of Section 4.3 is a major revision of the sequent calculus FL_{Seq} of Gabbay and Cheney [41], which in turn is based on prior work on a natural deduction calculus called Fresh Logic (FL) by Gabbay [40]. Caires and Cardelli [14] developed a sequent calculus for reasoning about concurrent processes that may calculate with or hide names, using a И-quantifier. In this system, information about freshness is maintained in a side-context consisting of freshness assertions, but freshness formulas are not present; their system uses an auxiliary concept called "free terms" similar to slices. Schöpp and Stark's system [114] uses *bunched contexts* [98] which generalize the structured contexts Σ used here. In their system, the *resources* of bunched logic are identified with the *names* of nominal logic. Contexts may be constructed using either Γ, Γ', indicating that Γ and Γ' refer to separate name-spaces, or $\Gamma; \Gamma'$, which permits Γ and Γ' to refer to overlapping name-spaces. Our contexts Σ, x and $\Sigma \# \mathsf{a}$ correspond to their contexts $\Gamma; x$ and Γ, a respectively.

Chapter 5

Model Theory

A theory has only the alternative of being right or wrong. A model has a third possibility—it may be right but irrelevant.

—Manfred Eigen

In this chapter we prove several important results about nominal set models of NL. First, we address the issues of soundness and completeness. We review the problems with completeness with respect to finite support nominal sets, and then show that NL is both sound and complete with respect to ideal-supported nominal sets. Then we study theories of NL that have *nominal term models* (or *Herbrand models*), that is, models constructed from constants, function symbols, names, and abstractions. We generalize the idea of a *universal theory* to *nominal-universal theory* and prove a generalization of Herbrand's Theorem, that term models always exist for such theories. This is an important ingredient of the semantics for Horn clause nominal logic programming, which is the focus of the next chapter.

5.1 The Incompleteness of Finite-Support Models

Pitts introduced a semantics based on nominal sets in which all elements are finitely supported. This class of models is equivalent (in our notation) to $\mathcal{P}_{<\omega}(\mathbb{A})$-supported nominal sets, since $\mathcal{P}_{<\omega}(\mathbb{A})$ consists of precisely the finite subsets of $\mathbb{A}$. We refer to nominal sets in $\mathbf{Nom}(\mathcal{P}_{<\omega}(\mathbb{A}))$ as *finite-support* nominal sets.

The finite-support semantics is intuitively appealing because abstract syntax trees are usually finite objects that mention only finitely many names. But several desirable properties which hold for first-order logic and structures fail for nominal logic and finite-support nominal structures. For example, the compactness theorem fails:

Remark 5.1.1 (Noncompactness). There exists a set of formulas such that every finite subset has a finite-support model but the whole has no finite-support model.

Suppose $\{\mathsf{a}_i^\nu\}_{i\in\omega}$ is an infinite sequence of names of the same sort ν. Consider $\Gamma = \{\neg(\mathsf{a}_i \# x^\delta) \mid i \in \omega\}$. For any finite subset Γ_0 of Γ, let n be the largest index of an a_i occurring in Γ_0. Let $\mathcal{M}$ be the finite-support model in which δ is interpreted as the set of finite sequences of names. Then each formula $\neg(\mathsf{a}_i \# x) \in \Gamma_0$ is satisfied by taking $x = [\mathsf{a}_1, \dots, \mathsf{a}_n]$. Nevertheless, Γ is not satisfiable in any finite-support model, since it asserts that there is an x with infinite support.

Indeed, the closely related property of *completeness* fails for finite-support models as well. Completeness implies compactness, so Remark 5.1.1 provides a counterexample to completeness. In fact, Pitts gave a direct counterexample to completeness in [108, Sec. 5, Example 4], in which the theory Γ is finite and closed.

The failure of the completeness theorem is a serious problem for any logic. There appear to be several reasonable approaches to solving this problem. We could search for an alternative logic which is complete with respect to finite support nominal sets; such a logic is unlikely to be first-order. Or we could introduce a new semantics with respect to which nominal logic is complete. This is the approach we have taken.

In the next two sections we show that nominal logic actually is sound and complete with respect to ideal-supported nominal set models.

5.2 Soundness

As a first step, we show

Proposition 5.2.1. *If* $\mathsf{a} \in \Sigma$, $\rho : \Sigma$, *and* $\Sigma|_\mathsf{a} \vdash t : \sigma$ *then* $\Sigma : \rho \vDash \mathsf{a} \# t$.

Proof. We first show that $\rho(\mathsf{a}) \# \rho(x)$ for each $x \in \Sigma|_\mathsf{a}$. Proof by induction on the construction of $\Sigma|_\mathsf{a}$. For $\Sigma'\#\mathsf{a}|_\mathsf{a} = \Sigma'$, if $x \in \Sigma'$ then $\rho(\mathsf{a}) \# \rho(x)$ since $\rho : \Sigma'\#\mathsf{a}$. For $\Sigma', y|_\mathsf{a} = \Sigma'|_\mathsf{a}$, if $x \in \Sigma|_\mathsf{a}$ then $x \in \Sigma'|_\mathsf{a}$ so proceed by induction. Similarly for the case $\Sigma'\#\mathsf{b}|_\mathsf{a} = \Sigma'|_\mathsf{a}\#\mathsf{b}$.

Now we show that $\Sigma : \rho \vDash \mathsf{a} \# t$ by induction on the derivation of $\Sigma|_\mathsf{a} \vdash t : \sigma$. If t is a name-constant b, then it is not a since $\mathsf{a} \notin \Sigma|_\mathsf{a}$, so $\rho(\mathsf{a}) \# \rho(\mathsf{b})$. If t is a variable $x \in \Sigma|_\mathsf{a}$, then by the above argument $\mathsf{a} \# \rho(x)$. The cases for $t = \langle\rangle$, $t = f(t')$, and $t = \langle t_1, t_2\rangle$ are straightforward. □

Remark 5.2.2. To state the soundness and completeness properties in full generality we need to use infinite contexts Σ, since an infinite set of formulas may mention infinitely many names. This is for bookkeeping reasons only, so that we can talk about infinite sets of formulas. To work with infinite contexts, it is necessary to generalize the universe of nominal terms to allow infinite terms supported by some support ideal; however, this is straightforward. It is also straightforward to generalize the other definitions involving contexts to the infinite case.

For example, the context implicitly used in the compactness counterexample is $\Sigma = (\mathsf{a}_1\#\mathsf{a}_2\#\cdots), x$. If the context $x\#\mathsf{a}_1\#\mathsf{a}_2\#\cdots$ had been used instead, then Γ would not have been finitely satisfiable since the context guarantees that a_i is fresh for each x.

Definition 5.2.3. *Let Γ be a set of formulas (possibly infinite), and ϕ a formula, both well-formed with respect to a (possibly infinite) context Σ. We write $\Sigma : \Gamma \vdash \phi$ when $\Sigma : \Gamma_0 \Rightarrow \phi$ for some finite subset $\Gamma_0 \subseteq \Gamma$.*

We write $\Sigma : \Gamma \vDash \phi$ to indicate that whenever $\Sigma, \mathcal{M} \vDash \Gamma$, we also have $\Sigma, \mathcal{M} \vDash \phi$.

We will first show that all the basic axioms concerning freshness and swapping are valid for nominal models.

Proposition 5.2.4 (S_1–$S_3, E_{\mathbf{1}}, E_f, E_\times, E_p$). *The swapping and equivariance axioms are valid in nominal models.*

Proof. The swapping axioms follow immediately from the definition of nominal sets. $E_{\mathbf{1}}, E_f, E_\times, E_p$ follow from the fact that unit, function symbols, pairing, and relation symbols are interpreted as equivariant constants, function symbols, and relations respectively. In particular, the built-in function symbols for swapping, abstraction, freshness, and equality are interpreted as equivariant functions. □

Lemma 5.2.5 (F_1). *If $(a\ b) \in \mathbb{A}^{[2]}$, $a \# x$, and $b \# x$ then $(a\ b) \cdot x = x$.*

Proof. Suppose $a, b \#_X x$. Then $a, b \notin supp_X(x)$. By the definition of support, we have $(a\ b) \cdot_X x = x$. □

The following lemmas are immediate consequences of the fact that different name-sorts are interpreted as disjoint name-sets, and that $supp(a) = \{a\}$ for a a name, respectively.

Lemma 5.2.6 (F_2). *If A, B are distinct name-sets, $a \in A, b \in B$, then $a \# b$.*

Lemma 5.2.7 (F_{3a}, F_{3b}). *If A is a name-set and $a, b \in A$ then $a \# b \iff a \neq b$.*

Lemma 5.2.8 (A_1). *$\langle a \rangle x = \langle b \rangle y$ if and only if $a = b$ and $x = y$ or $a \# y$ and $x = (a\ b) \cdot y$.*

Lemma 5.2.9 (U). *If $x \in \mathbf{1}$ then $x = \langle\rangle$.*

Proposition 5.2.10. *If $\Sigma : \Gamma \Rightarrow \Delta$ then $\Sigma : \Gamma \vDash \bigvee \Delta$.*

Proof. It will suffice to show that all of the proof rules and axioms of Figures 4.4–4.7 are valid with respect to $\vDash$. The cases for the first-order and equational logic rules are standard. We will use Theorem 4.2.19 and Theorem 4.2.17 several times without explicit reference.

(Ax): We have already shown that all the basic equational and freshness axioms are valid. It is an easy matter to show that if $\bigwedge P \supset \bigvee Q$ is a valid formula then

$$\frac{\Sigma : \Gamma, \bigwedge P, Q_1 \Rightarrow \Delta \quad \cdots \quad \Sigma : \Gamma, \bigwedge P, Q_n \Rightarrow \Delta}{\Sigma : \Gamma, \bigwedge P \Rightarrow \Delta}$$

is a valid inference rule.

(A_2): Suppose the derivation is of the form:

$$\frac{\Sigma \vdash t : \langle\nu\rangle\sigma \quad \Sigma\#\mathsf{a}, x : \Gamma, t \approx \langle\mathsf{a}\rangle x \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \Delta} A_2$$

Assume $\Sigma : \rho \vDash \Gamma$. By Proposition 4.2.6 we must have $\rho(t) \in [\![\langle\nu\rangle\sigma]\!]$. Choose fresh $v \in [\![\nu]\!] - \mathit{supp}(\rho)$, and choose $v' \in [\![\sigma]\!]$ such that $\rho(t) = \langle v\rangle v'$. Let $\rho' = \rho[\mathsf{a} \mapsto v, x \mapsto v']$; since v is fresh for ρ, we have $\rho' : \Sigma\#\mathsf{a}, x$, and moreover, $\Sigma\#\mathsf{a}, x : \rho' \vDash t \approx \langle\mathsf{a}\rangle x$. Hence $\Sigma\#\mathsf{a}, x : \rho' \vDash \Gamma, t \approx \langle\mathsf{a}\rangle x$ so by induction $\Sigma\#\mathsf{a}, x : \rho' \vDash \bigvee\Delta$. But since a and x are fresh for Δ, we also have $\Sigma : \rho \vDash \bigvee\Delta$.

(P): The proof of the soundness of the surjectivity rule for pairing is standard and straightforward.

$(\Sigma\#)$: Suppose we have a derivation of the form:

$$\frac{\Sigma|_{\mathsf{a}} \vdash t : \sigma \quad \Sigma : \Gamma, \mathsf{a} \# t \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \Delta} \Sigma\#$$

Let $\Sigma : \rho \vDash \Gamma$ be given. Then by Proposition 5.2.1, we know $\Sigma : \rho \vDash \mathsf{a} \# t$, so $\Sigma : \rho \vDash \Gamma, \mathsf{a} \# t$ and by the induction hypothesis we can conclude $\Sigma : \rho \vDash \bigvee\Delta$.

(F): Suppose we have a derivation of the form:

$$\frac{\Sigma\#\mathsf{a}^\nu : \Gamma \Rightarrow \Delta}{\Sigma : \Gamma \Rightarrow \Delta} F$$

where $\mathsf{a} \notin \Sigma$. We need to show that whenever $\Sigma : \rho \vDash \Gamma$ we have $\Sigma : \rho \vDash \bigvee\Delta$. Let $\Sigma : \rho \vDash \Gamma$ be given. Choose $v \in [\![\nu]\!]$ fresh for ρ. Then we have $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \Gamma$. By induction, we know that $\Gamma \vDash \bigvee\Delta$. Since $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \Gamma$ we can conclude $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \bigvee\Delta$ also, and since a is fresh we can also conclude $\Sigma : \rho \vDash \bigvee\Delta$.

$(\text{И}L)$: Suppose we have a derivation

$$\frac{\Sigma\#\mathsf{a} : \Gamma, \phi \Rightarrow \Delta}{\Sigma : \Gamma, \text{И}\mathsf{a}^\nu.\phi \Rightarrow \Delta}$$

where $\mathsf{a} \# \Gamma, \Delta$. Let ρ be given and assume $\Sigma : \rho \vDash \Gamma, \text{И}\mathsf{a}^\nu.\phi$. Choose fresh $v \in [\![\nu]\!] - \mathit{supp}(\rho)$. Then by the definition of $\vDash$, $\Sigma\#\mathsf{a} : \rho[\mathsf{a} \mapsto v] \vDash \Gamma, \phi$, so $\Sigma\#\mathsf{a}[\mathsf{a} \mapsto v] : \rho \vDash \bigvee\Delta$ by induction. Since a is fresh for Δ, we have $\Sigma : \rho \vDash \bigvee\Delta$.

$(\text{И}R)$: Suppose we have a derivation

$$\frac{\Sigma\#\mathsf{a} : \Gamma \Rightarrow \phi, \Delta}{\Sigma : \Gamma \Rightarrow \text{И}\mathsf{a}^\nu.\phi, \Delta}$$

where $\mathsf{a} \notin \Sigma, \Gamma, \Delta$. Let ρ be given and assume $\Sigma : \rho \vDash \Gamma$. Choose $v \in [\![\nu]\!] - \mathit{supp}(\rho)$ fresh, and let $\rho' = \rho[\mathsf{a} \mapsto v]$. Then we have $\Sigma\#\mathsf{a} : \rho' \vDash \Gamma$, hence by induction we know $\Sigma\#\mathsf{a} : \rho' \vDash \phi \vee \bigvee\Delta$. There are two cases depending on which disjunct is satisfied by ρ'. If $\Sigma\#\mathsf{a} : \rho' \vDash \bigvee\Delta$ then clearly $\Sigma : \rho \vDash \text{И}\mathsf{a}.\phi \vee \bigvee\Delta$ since a is fresh for $\text{И}\mathsf{a}.\phi \vee \bigvee\Delta$ and $v \# \rho$. Otherwise, $\Sigma\#\mathsf{a} : \rho' \vDash \phi$. Since a is fresh for Σ and v fresh for ρ, we may conclude that $\Sigma : \rho \vDash \text{И}\mathsf{a}.\phi$, and clearly also $\Sigma : \rho \vDash \text{И}\mathsf{a}.\phi \vee \bigvee\Delta$. Hence, in either case, we have $\Sigma : \rho \vDash \text{И}\mathsf{a}.\phi \vee \bigvee\Delta$ as desired. □

Theorem 5.2.11 (Soundness). *If $\Gamma \vdash \phi$ then $\Gamma \vDash \phi$.*

Proof. Let Γ_0 be a finite subset of Γ with $\Gamma_0 \Rightarrow \phi$ derivable. By the previous lemma, $\Gamma_0 \vDash \phi$, so $\Gamma \vDash \phi$. □

5.3 Completeness

Following the usual approach to proving completeness, we prove that every consistent theory has a nominal model.

Theorem 5.3.1 (Henkin's Theorem for NL). *If Γ is consistent then it has a nominal model.*

Proof. We construct (using the Axiom of Choice) a maximal consistent theory Γ^* and language $\mathcal{L}^*$ extending Γ and $\mathcal{L}$ in the usual way such that every existential formula $\exists x.\phi(x) \in \Gamma^*$ has a witnessing constant $c^* \in \mathcal{L}^*$ such that $\phi(c^*) \in \Gamma^*$. Given such a maximal consistent set we will show how to define a nominal model, by showing that the internal notion of support defined by Γ^* generates a support ideal.

Let $\mathcal{L}^*$ be the extended language used in constructing Γ^*. Note that $\mathcal{L}^*$ may contain added constants for which no equivariance axiom is assumed; their interpretations in the model might therefore not be equivariant and could even be infinitely supported.[1] We define the sort-interpretations $[\![\sigma]\!]$ as $\mathbb{T}(\sigma)/{\equiv_\sigma}$, where $\mathbb{T}(\sigma) = \{t : \sigma \mid FV(t) = \varnothing\}$ is the set of well-formed ground terms of sort σ in $\mathcal{L}^*$, and $\equiv_\sigma$: $\mathbb{T}(\sigma) \times \mathbb{T}(\sigma)$ is defined by

$$t \equiv_\sigma u \iff (t \approx u) \in \Gamma^* \ . \tag{5.1}$$

We write $[t]$ for the equivalence class of t under $\equiv_\sigma$, for $t : \sigma$.

We wish to endow $\mathbb{T}(\sigma)$ with an appropriate nominal set structure. To do so, we must identify an appropriate set of names. Note that $\mathcal{L}$ is defined in terms of the names $\mathbb{A}$ used for name-constants and variables in $\mathcal{L}$-formulas. However, $\mathcal{L}^*$ may include new constants of name-sort, and there is also the $\equiv_\sigma$ to contend with. Therefore, we will define a new set of names $\mathbb{A}^*$ for the language $\mathcal{L}^*$ and term sets $\mathbb{T}(\sigma)/{\equiv_\sigma}$.

We define $\mathbb{A}^*_\nu = \{[a] \mid a \in \mathbb{T}(\nu)\} = \mathbb{T}(\nu)/{\equiv_\nu} = [\![\nu]\!]$, and $\mathbb{A}^*$ to be the union of the $\mathbb{A}^*_\nu$. We define a swapping action on each $\mathbb{T}(\sigma)$ as follows:

$$([a]\ [b]) \cdot_{[\![\sigma]\!]} [t] = [(a\ b) \cdot t] \ . \tag{5.2}$$

For this function to be well-defined, we need to show that swapping is a $\equiv_\sigma$-congruence for each sort σ. That is, if $a \equiv_\nu a', b \equiv_\nu b', t \equiv_\sigma t'$, then $(a\ b) \cdot t \equiv_\sigma (a'\ b') \cdot t'$. This is tedious but straightforward, since the congruence laws for the swapping function symbol are valid in nominal logic and so all the needed

[1] This is why completeness cannot be shown using finite support models.

axiom instances must be in Γ^*. Once this is done, it is not difficult to verify that each $\mathbb{T}(\sigma)/\equiv_\sigma$ is a $\mathbb{G}^*$-set, where $\mathbb{G}^*$ is the name-group of all finite sort-respecting permutations on $\mathbb{A}^*$. Note that, for the new non-equivariant constants of $\mathcal{L}^*$, we may have $[(a\ b) \cdot c] \neq [c]$.

To show that each $[\![\sigma]\!]$ is a nominal set, we need to show that there is a support ideal $\mathcal{I}$ containing supports for all elements of $M = \bigcup_\sigma [\![\sigma]\!]$. We define

$$\mathit{supp}^*([t]) = \{[a] \in \mathbb{A}^* \mid \neg(a \# t) \in \Gamma^*\} \ .$$

That is, $\mathit{supp}^*(x)$ is the set of names that are not fresh for the term representing t according to Γ^*.

We now show that for every σ and every $x \in [\![\sigma]\!]$, $\mathit{supp}^*(x) \lhd x$. Suppose $[t] \in [\![\sigma]\!]$. Let compatible $[a], [b] \notin \mathit{supp}^*([t])$ be given. Then $\neg(a \# t), \neg(b \# t) \notin \Gamma^*$ so $a \# t, b \# t \in \Gamma^*$ since Γ^* is maximal. Moreover, $a \# t \wedge b \# t \supset (a\ b) \cdot t \approx t \in \Gamma^*$ since it is an axiom of NL. Therefore, we must have $(a\ b) \cdot t \approx t \in \Gamma^*$, hence

$$[t] = [(a\ b) \cdot t] = ([a]\ [b]) \cdot [t]$$

as desired. Therefore, $\mathit{supp}^*([t]) \lhd [t]$. Since $[t] \in [\![\sigma]\!]$ and σ were arbitrary, this shows that every element of every domain $[\![\sigma]\!]$ is supported in $\mathcal{I}$.

Now we define

$$\mathcal{I} = \{S \mid \exists x \in M.\ S \subseteq \mathit{supp}^*(x)\}$$

Obviously, $\mathcal{I}$ contains a support for every $x \in M$. To show that $\mathcal{I}$ is a support ideal, we need to show that the properties (1–4) defining a support ideal hold:

1. If $T \subseteq S \in \mathcal{I}$ then for some x, $T \subseteq S \subseteq \mathit{supp}^*(x)$, so $T \in \mathcal{I}$.

2. If $S, T \in \mathcal{I}$ then $S \subseteq \mathit{supp}^*([x])$ and $T \subseteq \mathit{supp}^*([y])$. Then

$$S \cup T \subseteq \mathit{supp}^*([x]) \cup \mathit{supp}^*([y]) = \mathit{supp}^*([x],[y]) = \mathit{supp}^*([\langle x, y\rangle])$$

   so $S \cup T \in \mathcal{I}$. Here we use various properties of freshness, swapping, and pairing, including surjectivity.

3. For any name $[a]$, we have $\mathit{supp}^*([a]) = \{[b] \mid \neg([b] \# [a]) \in \Gamma^*\}$. By the axioms F_{3a}, F_{3b}, we know $\neg(a \# a) \in \Gamma^*$ and $a \# b \vee a \approx b \in \Gamma^*$. This shows that $[a] \in \mathit{supp}^*([a])$ and that if $[b] \in \mathit{supp}^*([a])$ then $[a] \approx [b]$ since otherwise we would have $a \# b$ and $\neg(a \# b) \in \Gamma^*$, a contradiction. So $\{[a]\} \in \mathcal{I}$.

4. We need to show that no name-set $\mathbb{A}^*_\nu = [\![\nu]\!]$ is in $\mathcal{I}$, that is, for any name-sort ν and set $S \in \mathcal{I}$ we can find a name $a \in [\![\nu]\!] - S$. If $S \in \mathcal{I}$ then $S \subseteq \mathit{supp}^*([x])$ for some $[x] \in M$. Since Γ^* is closed under applications of the rules (F) and $(\Sigma\#)$, for any $[x] \in M$, there must be an $[\mathsf{a}] \in [\![\nu]\!]$ such that $(\mathsf{a} \# x) \in \Gamma^*$. Therefore, $\neg(\mathsf{a} \# x) \notin \Gamma^*$ for each i, and so $[\mathsf{a}] \notin \mathit{supp}^*([x])$, hence $[\mathsf{a}] \notin S$.

Thus, we may consider $[\![\sigma]\!]$ as a nominal set with the swapping action (5.2). We may therefore define a nominal set model $\mathcal{M}$ with sort interpretation as above, $[\![c]\!] = [c]$, $[\![f(t)]\!] = [f]([\![t]\!])$, where $[f]([t]) = [f(t)]$, and $[\![p]\!] = \{[t] \mid p(t) \in \Gamma^*\}$. It is easy to verify that the interpretations of the constant, function, and relation symbols $c, f, p \in \mathcal{L}$ are equivariant constants, functions, and relations respectively, using the equivariance rules. This shows that $\mathcal{M}$ is a nominal structure; moreover, it is a model of Γ^* and so also Γ.

Consequently, $\mathcal{M}$ is a nominal set model of Γ supported by $\mathcal{I}$. □

Theorem 5.3.2 (Completeness). *If* $\Gamma \vDash \phi$ *then* $\Gamma \vdash \phi$.

Proof. Suppose $\Gamma \vDash \phi$. Then $\Gamma, \neg\phi$ is not satisfiable by Theorem 5.3.1, so inconsistent. Consequently $\Gamma, \neg\phi \vdash \phi$. Since $\Gamma, \phi \vdash \phi$ obviously, we get $\Gamma \vdash \phi$ by using the law of excluded middle. □

Corollary 5.3.3 (Compactness). *If* Γ *is finitely satisfiable then* Γ *is satisfiable.*

Corollary 5.3.4 (Löwenheim-Skolem). *If a set of nominal logic formulas* Γ *over a language* $\mathcal{L}$ *is satisfiable, then it has models of all infinite cardinalities* $\kappa \geq max(|\Gamma|, |L|)$.

5.4 Herbrand's Theorem

First-order (sorted) logic satisfies an important property called *Herbrand's Theorem*: every collection of *universal* formulas has a model built up out of *closed terms*. Term models are interesting because logic programming takes place in the universe of terms. Because terms are finite, term models have finite support so we do not have to consider arbitrary support ideals. In many applications of nominal logic, the objects we wish to reason about are terms, and so term models are of primary interest.

However, Herbrand's Theorem need not be true of an arbitrary theory, and in particular, it is not true of Pitts' formulation of NL since its Hilbert-style first-order axiomatization includes existential formulas, in particular, in the freshness axiom $F4$

$$(F4) \quad \forall \vec{x}.\exists a.a \# \vec{x}$$

The existence of closed equivariant terms of name-sorts ν lead to inconsistency, because

$$\forall a^{\nu}.a \# t$$

is provable for any closed term t. If $t : \nu$ then instantiating a to t yields $t \# t$, a contradiction.

In NL$^{\Rightarrow}$, we fixed this problem by adding non-equivariant constants for names. Only constants of non-name sort are assumed equivariant. Since name-constants are closed terms, it is no longer the case that closed terms have empty support, so $\forall a^{\nu}.a \# t$ is not valid for all closed terms t. Furthermore, a term may be closed and equivariant while still mentioning names: for example, $l(\langle \mathsf{a}\rangle \mathsf{a})$ is equivariant since a is bound. In NL, on the other hand, there is no closed term for this equivariant value.

In this section, we will interpret nominal terms in the universe of nominal terms, using semantic swapping and abstraction for syntactic swapping and abstraction respectively. That is, we interpret

$$[\![f]\!] \quad = \quad x \mapsto f(x) : \mathbb{NT}(\sigma) \to \mathbb{NT}(\sigma')$$

provided $f : \sigma \to \sigma'$, and interpret unit, pairing, swapping, and abstraction as the respective semantic versions in $\mathbb{NT}$. This is in contrast to the previous section, where in the proof of Henkin's Theorem we interpreted swapping and abstraction in a nominal structure consisting of first-order terms from $\mathbb{T}$ quotiented by an equivalence relation. It would not have been possible to use nominal term abstraction and swapping directly there because a general consistent theory Γ may identify distinct nominal terms such as $f(x)$ and $\langle a\rangle x$. However, in this section we will consider only *nominal-universal theories*, for which nominal term models will be shown to be complete. Therefore, we can use nominal terms in a "deep" way to simplify the proof: the equivalence relation for identifying terms up to abstraction is still there, but we have already done all the work necessary to avoid talking about it explicitly.

We write $\mathbb{NT}(\sigma) = \{[\![t]\!] \mid \cdot \vdash t : \sigma\}$ for the set of all nominal term interpretations of well-formed terms of sort σ. Note that the recursive definitions of swapping, freshness, and equality of Section 3.5 can be used to calculate with nominal terms via their first-order representations. We will use this fact in proofs.

In the rest of this section, let $\mathcal{L}$ be a fixed language.

Definition 5.4.1. *The* Herbrand sort interpretation *over $\mathcal{L}$ is the sort interpretation generated by*

$$[\![\delta]\!] = \mathbb{NT}(\delta)$$

and the Herbrand universe *is the universe of this interpretation.*

Definition 5.4.2. *The* nominal Herbrand term interpretation *over $\mathcal{L}$ is the term interpretation generated by*

$$[\![f]\!](x) = f(x)$$

Definition 5.4.3. *A* Herbrand structure *is a structure over the Herbrand universe and term interpretation.*

Proposition 5.4.4. *Any Herbrand structure is a model of NL.*

Proof. Any Herbrand structure is a (finite-support) nominal structure by assumption, so this is just a consequence of the soundness of NL for nominal structures. $\square$

Intuitively, once a language $\mathcal{L}$ is given, the definition of swapping, the behavior of equality and freshness, and the Herbrand sort and term interpretations are fixed. The remaining information about a given model is captured by the set of atomic formulas true in it.

Recall that formulas $t \approx u, t \mathrel{\#} u$ are called constraints. Other atomic formulas $p(\vec{t})$ are called basic formulas.

Definition 5.4.5. *The* Herbrand base *over $\mathcal{L}$, written $B_{\mathcal{L}}$, consists of all basic $\mathcal{L}$-formulas. An* Herbrand interpretation *is a subset of $B_{\mathcal{L}}$.*

Proposition 5.4.6. *Any Herbrand interpretation B generates a unique Herbrand model $\mathcal{H} \vDash B$. Conversely, any Herbrand model $\mathcal{H}$ is generated by a unique Herbrand interpretation.*

Proof. Let B be an Herbrand interpretation. Let $\mathcal{H}$ be the structure over the Herbrand universe and term interpretation and with $[\![p]\!]$ defined as $\{t \mid p(t) \in B\}$. Since B is equivariant, $[\![p]\!]$ is also equivariant. Thus, $\mathcal{H}$ is a Herbrand structure; it is a model of B since every element of B is valid in $\mathcal{H}$.

Let $\mathcal{H}$ be an Herbrand model. Let $B = \{A \mid \mathcal{H} \vDash A\}$ be the set of all basic formulas true in $\mathcal{H}$. Clearly B is a Herbrand base, and it generates $\mathcal{H}$ using the construction in the previous paragraph. □

Corollary 5.4.7. *Nominal logic is consistent.*

Proof. By the Proposition 5.4.4, any Herbrand structure is a model of NL. Herbrand structures exist: for example, $\varnothing$ generates a nominal structure by Proposition 5.4.6. □

The main result of this section is a nominal version of Herbrand's theorem: that every theory of a particular form (in this case, every *nominal-universal theory*) has an Herbrand model.

Definition 5.4.8. *A* nominal-universal *(or* И∀-*)* formula *is a formula of the form*

$$\phi ::= \psi \mid C \supset \phi \mid \text{И}\mathsf{a}.\phi \mid \forall x.\phi$$

where ψ is a quantifier-free formula mentioning only basic formulas, and C is a constraint. Thus, a И∀*-formula consists of a sequence of* И/∀*-quantifiers and $\approx$/#-constraints, followed by a quantifier-free body not mentioning # or $\approx$.*

A nominal-universal theory *(or* И∀*-theory) is a set Γ of closed* И∀*-formulas.*

The reason that constraints are allowed only in restricted positions is to prevent equality and freshness from being redefined. If positive freshness or equality formulas were permitted then theories like $\Gamma = \{\forall x^\nu, y^\nu . y \mathrel{\#} f(x)\}$ would be allowed, which are not satisfiable in an Herbrand universe because $\neg(\mathsf{a} \mathrel{\#} f(\mathsf{a}))$ where $f(\mathsf{a})$ is an uninterpreted term. Also, theories with unsatisfiable formulas like И$\mathsf{a}.\mathsf{a} \mathrel{\#} \mathsf{a}$ would be allowed. This is a familiar problem with incorporating equational reasoning into logic programming.

We now show several technical lemmas concerning Herbrand models, substitution, and the relationship between Herbrand and general models.

Lemma 5.4.9. *If $\rho : \Sigma, x$ and $u : \sigma$ is ground then $\rho[x \mapsto [\![u]\!]](t) = \rho(t\{u/x\})$.*

If $\mathcal{H}$ is a nominal Herbrand model and ρ an $\mathcal{H}$-valuation and t is ground then $\Sigma, x : \rho[x \mapsto [\![t]\!]] \vDash \phi$ iff $\Sigma : \rho \vDash \phi\{t/x\}$.

Proof. Straightforward induction on the definition of $\mathcal{T}[\![t]\!]\rho$ and $\Sigma : \rho \vDash \phi$. □

Lemma 5.4.10. *If $\mathcal{H}$ is a nominal Herbrand model and ρ an $\mathcal{H}$-valuation then $\Sigma : \rho \vDash \forall x^{\sigma}.\phi$ iff $\Sigma : \rho \vDash \phi\{t/x\}$ for every ground $t : \sigma$.*

Proof. If $\rho \vDash \forall x.\phi$, then $\rho[x \mapsto v] \vDash \phi$ for all $v \in [\![\sigma]\!]$. Let ground $t : \sigma$ be given; then $\rho[x \mapsto [\![t]\!]] \vDash \phi$. By Lemma 5.4.9, $\rho \vDash \phi\{t/x\}$. The choice of t was arbitrary, so $\rho \vDash \phi\{t/x\}$ for every ground term t.

If $\rho \vDash \phi\{t/x\}$ for every ground term $t : \sigma$, then let $v \in [\![\sigma]\!]$ be given. By definition there is a ground term $t : \sigma$ such that $v = [\![t]\!]$; and by assumption, $\rho \vDash \phi\{t/x\}$. Then by Lemma 5.4.9, $\rho[x \mapsto [\![t]\!]] \vDash \phi$ so $\rho[x \mapsto v] \vDash \phi$. The choice of v was arbitrary, so $\rho[x \mapsto v] \vDash \phi$ for each $v \in [\![\sigma]\!]$, and so $\rho \vDash \forall x.\phi$. □

The following fact is an important consequence of the previous lemma.

Corollary 5.4.11. *Let $\phi = \text{И}\vec{\mathsf{a}}.\forall\vec{x}.\psi$ be a closed formula, where $x_i : \sigma_i$, and let $\mathcal{H}$ be an Herbrand model. Then $\mathcal{H} \vDash \phi$ if and only if $\mathcal{H} \vDash \psi\{t_1/x_1\}\cdots\{t_m/x_m\}$ for all ground $t_i : \sigma_i$.*

The proof is a straightforward induction on the number of quantifiers of ϕ. We next show that any ground constraint valid in an Herbrand model $\mathcal{H}$ is valid in any model of NL.

Lemma 5.4.12. *If $\mathcal{H}$ is an Herbrand model and C is a ground constraint and $\mathcal{H} \vDash C$ then $\mathcal{M} \vDash C$ for any structure $\mathcal{M}$.*

Proof. Note that $\mathcal{H} \vDash a \# u$ iff $[\![a]\!] \#_{\mathbb{NT}} [\![u]\!]$ iff $fresh(a, u) = true$ and $\mathcal{H} \vDash t \approx u$ iff $[\![t]\!] = [\![u]\!]$ iff $eq(t, u)$. By the soundness of NL, it suffices to show that if $fresh(a, t) = true$ or $eq(t, u) = true$ then $\Rightarrow a \# t$ or $\Rightarrow t \approx u$ respectively is derivable in NL, since then $\mathcal{M} \vDash a \# t$ or $t \approx u$ respectively for any model $\mathcal{M}$.

For formulas $\mathsf{a} \# u$, the proof is by induction on the definition of $fresh$.

If $t = \mathsf{b}$ and $\mathsf{a} \neq \mathsf{b}$, i.e., a and b are distinct name-constants, then

$$\frac{\Sigma|_{\mathsf{a}} \vdash \mathsf{b} : \nu \quad \Sigma : \mathsf{a} \# \mathsf{b} \Rightarrow \mathsf{a} \# \mathsf{b}}{\Sigma : \cdot \Rightarrow \mathsf{a} \# \mathsf{b}}\ \Sigma\#$$

If $t = \langle\rangle$, then

$$\frac{\Sigma|_{\mathsf{a}} \vdash c : \sigma \quad \Sigma : \mathsf{a} \# \langle\rangle \Rightarrow \mathsf{a} \# \langle\rangle}{\Sigma : \cdot \Rightarrow \mathsf{a} \# \langle\rangle}\ \Sigma\#$$

If $t = f(t')$, then we must have $\mathcal{H} \vDash \mathsf{a} \# t'$, so $\Sigma : \cdot \Rightarrow \mathsf{a} \# t'$. Then we may derive

$$\cfrac{\cfrac{\Sigma\#\mathsf{b}|_{\mathsf{b}} \vdash f(t'), t' : \vec{\sigma} \quad \Sigma\#\mathsf{b} : \mathsf{b} \# f(t'), t' \Rightarrow \mathsf{a} \# f(t')}{\Sigma\#\mathsf{b} : \cdot \Rightarrow \mathsf{a} \# f(t')}\ \Sigma\#^2}{\Sigma : \cdot \Rightarrow \mathsf{a} \# f(t')}\ F$$

where $\Sigma\#\mathsf{b} : \mathsf{b} \# f(t'), t' \Rightarrow \mathsf{a} \# f(t')$ follows using F_1, the derivations of $\Rightarrow \mathsf{a} \# u$, cut, and equivariance.

If $t = \langle t_1, t_2 \rangle$, then we must have $\mathcal{H} \vDash \mathsf{a} \# t_1, t_2$, so $\Sigma : \cdot \Rightarrow \mathsf{a} \# t_1, t_2$. Then we may derive

$$\dfrac{\dfrac{\Sigma\#\mathsf{b}|_{\mathsf{b}} \vdash \langle t_1, t_2 \rangle, t_1, t_2 : \vec{\sigma} \quad \Sigma\#\mathsf{b} : \mathsf{b} \# \langle t_1, t_2 \rangle, t_1, t_2 \Rightarrow \mathsf{a} \# \langle t_1, t_2 \rangle}{\Sigma\#\mathsf{b} : \cdot \Rightarrow \mathsf{a} \# \langle t_1, t_2 \rangle}\ \Sigma\#^3}{\Sigma : \cdot \Rightarrow \mathsf{a} \# \langle t_1, t_2 \rangle}\ F$$

where $\Sigma\#\mathsf{b} : \mathsf{b} \# \langle t_1, t_2 \rangle, t_1, t_2 \Rightarrow \mathsf{a} \# \langle t_1, t_2 \rangle$ follows using F_1, the derivations of $\Rightarrow \mathsf{a} \# t_i$, cut, and equivariance.

Finally, if $t = \langle \mathsf{b} \rangle t'$ then there are two cases. If $\mathsf{b} = \mathsf{a}$ then we have

$$\dfrac{\dfrac{\Sigma\#\mathsf{b}|_{\mathsf{b}} \vdash \mathsf{a}, t, \langle \mathsf{a} \rangle t : \vec{\sigma} \quad \Sigma\#\mathsf{b} : \mathsf{b} \# \mathsf{a}, t, \langle \mathsf{a} \rangle t \Rightarrow \mathsf{a} \# \langle \mathsf{a} \rangle t}{\Sigma\#\mathsf{b} : \cdot \Rightarrow \mathsf{a} \# \langle \mathsf{a} \rangle t}\ \Sigma\#}{\Sigma : \cdot \Rightarrow \mathsf{a} \# \langle \mathsf{a} \rangle t}\ F$$

where $\Sigma\#\mathsf{b} : \mathsf{b} \# \mathsf{a}, t, \langle \mathsf{a} \rangle t \Rightarrow \mathsf{a} \# \langle \mathsf{a} \rangle t$ follows since $\mathsf{b} \# \langle \mathsf{a} \rangle t \iff \mathsf{a} \# (\mathsf{b}\ \mathsf{a}) \cdot \langle \mathsf{a} \rangle t$ and $(\mathsf{b}\ \mathsf{a}) \cdot \langle \mathsf{a} \rangle t \approx \langle \mathsf{a} \rangle t$ since $\mathsf{a}, \mathsf{b} \# \langle \mathsf{a} \rangle t$. On the other hand, if $\mathsf{a} \not\approx \mathsf{b}$ then we must have $\mathcal{H} \vDash \mathsf{a} \# t$, so by induction $\Rightarrow \mathsf{a} \# t$. In addition we must have $\Rightarrow \mathsf{a} \# \mathsf{b}$ since $\mathsf{a} \neq \mathsf{b}$. This case is similar to that for an arbitrary function symbol.

For equations $t \approx u$ the proof is by induction on the definition of eq. However, all of the cases involving unit, pairing, name-constants, function symbols, or abstractions over the same name are easy, since in these cases equality is syntactic. The only interesting case is when $t = \langle \mathsf{a} \rangle t'$ and $u = \langle \mathsf{b} \rangle u'$ where $\mathsf{a} \neq \mathsf{b}$. In this case we know $fresh(\mathsf{a}, u') = true$ and $eq(t', u'(\mathsf{a} \leftrightarrow \mathsf{b})) = true$. Consequently by induction we can derive $\Sigma : \cdot \Rightarrow \mathsf{a} \# u'$ and $\Sigma : \cdot \Rightarrow t' \approx (\mathsf{a}\ \mathsf{b}) \cdot u'$. Using (A_1) we have

$$\dfrac{\Sigma : \langle \mathsf{a} \rangle t' \approx \langle \mathsf{b} \rangle u' \Rightarrow \langle \mathsf{a} \rangle t' \approx \langle \mathsf{b} \rangle u'}{\Sigma : \mathsf{a} \# u', t' \approx (\mathsf{a}\ \mathsf{b}) \cdot u' \Rightarrow \langle \mathsf{a} \rangle t' \approx \langle \mathsf{b} \rangle u'}\ A_1$$

Using cut twice we can therefore derive $\Sigma : \cdot \Rightarrow \langle \mathsf{a} \rangle t' \approx \langle \mathsf{b} \rangle u'$. □

Proposition 5.4.13. *Let $\mathcal{M}$ be a model and $\mathcal{H}$ be the Herbrand model generated by the base $B = \{A \mid \mathcal{M} \vDash A\}$, and let ϕ be closed. Then*

1. *If ϕ is quantifier-free, then $\mathcal{M} \vDash \phi$ if and only if $\mathcal{H} \vDash \phi$.*

2. *If ϕ is a $\text{И}\forall$-formula and $\mathcal{M} \vDash \phi$ then $\mathcal{H} \vDash \phi$.*

Proof. For (1), the proof is by induction on the structure of ϕ.

If ϕ is basic, then $\mathcal{M} \vDash \phi$ implies $\phi \in B$, so $\mathcal{H} \vDash \phi$. Similarly, if $\mathcal{H} \vDash \phi$ then $\phi \in B$ so $\mathcal{M} \vDash \phi$.

If ϕ is a conjunction $\phi_1 \wedge \phi_2$ then by induction $\mathcal{M} \vDash \phi_1, \phi_2$ so $\mathcal{H} \vDash \phi_1, \phi_2$ so $\mathcal{H} \vDash \phi$, and similarly for the reverse direction. If ϕ is a negation $\neg\psi$ then $\mathcal{M} \nvDash \psi$ so by induction $\mathcal{H} \nvDash \psi$ so $\mathcal{H} \vDash \neg\psi$, and similarly for the reverse direction. (This is why we need "if and only if"). The remaining propositional cases are similar.

For (2), there are four cases: ϕ is quantifier-free, of the form $C \supset \psi$, of the form $\forall x.\psi(x)$, or of the form $\text{И}\mathsf{a}.\psi(\mathsf{a})$, where ψ is $\text{И}\forall$.

- If ϕ is quantifier-free, then (1) applies.
- If $\phi = C \supset \psi$ is a constrained formula, then suppose $\mathcal{M} \vDash C \supset \psi$. Assume $\mathcal{H} \vDash C$. By Lemma 5.4.12, since C is a ground constraint and $\mathcal{H} \vDash C$, we must have $\mathcal{M} \vDash C$; hence, $\mathcal{M} \vDash \psi$, so by the induction hypothesis $\mathcal{H} \vDash \psi$.
- If $\phi = \forall x.\psi(x)$, then assume $\mathcal{M} \vDash \forall x.\psi(x)$. Then $\mathcal{M} \vDash \psi(x)$, so $\mathcal{M} \vDash \phi(t)$ for each closed term t. By induction, $\mathcal{H} \vDash \psi(t)$ for each closed term t. Hence by Lemma 5.4.10, $\mathcal{H} \vDash \forall x.\psi(x)$, or $\mathcal{H} \vDash \phi$.
- Finally, if $\phi = \text{И}\mathsf{a}.\psi(\mathsf{a})$, then assume $\mathcal{M} \vDash \text{И}\mathsf{a}.\psi(\mathsf{a})$. Then $\mathcal{M} \vDash \psi(\mathsf{a})$ for fresh a. We wish to show that $\mathcal{H} \vDash \text{И}\mathsf{a}.\psi(\mathsf{a})$. Let a be fresh. Then $\mathcal{M} \vDash \psi(\mathsf{a})$, so by induction $\mathcal{H} \vDash \psi(\mathsf{a})$. Since a was fresh, $\mathcal{H} \vDash \text{И}\mathsf{a}.\psi(\mathsf{a})$.

This completes the proof. $\square$

Definition 5.4.14 (Instances of И∀-formulas). *The set of* ground instances *$GI(\phi)$ of a closed И∀-formula ϕ is defined as follows:*

$$\begin{array}{rcl} GI(\psi) & = & \{g \cdot \psi \mid g \in \mathbb{G}\} \quad (\psi \textit{ quantifier-free}) \\ GI(C \supset \phi) & = & \begin{cases} GI(\phi) & (\vDash C) \\ \varnothing & (\vDash \neg C) \end{cases} \\ GI(\text{И}\mathsf{a}.\phi) & = & GI(\phi) \\ GI(\forall x^\sigma.\phi) & = & \bigcup_{t:\sigma} GI(\phi\{t/x\}) \end{array}$$

Note that in the case for $C \supset \phi$, C is always ground, so is either valid or unsatisfiable. We write $\Gamma \mathrel{=\!\!|\!\vDash} \Gamma'$ to indicate that $\Gamma \vDash \Gamma'$ and $\Gamma' \vDash \Gamma$.

Lemma 5.4.15. *If ϕ is a closed И∀-formula, then the set $GI(\phi)$ of all instances of ϕ satisfies $\phi \mathrel{=\!\!|\!\vDash} GI(\phi)$.*

Proof. The proof is by induction on the structure of ϕ. If ϕ is quantifier-free, then all of its instances are similar up to a permutation, and so all are equivalent formulas and their conjunction is equivalent to ϕ. If $\phi = C \supset \psi$ and C holds then $\phi = C \supset \psi \mathrel{=\!\!|\!\vDash} \psi \mathrel{=\!\!|\!\vDash} GI(\psi) = GI(\phi)$. If C does not hold, then $\phi \mathrel{=\!\!|\!\vDash} \top \mathrel{=\!\!|\!\vDash} \varnothing = GI(\phi)$. If $\phi = \text{И}\mathsf{a}.\psi$ then $\phi \mathrel{=\!\!|\!\vDash} \psi \mathrel{=\!\!|\!\vDash} GI(\psi) = GI(\phi)$. Finally, if $\phi = \forall x^\sigma.\psi$ then by Lemma 5.4.10, $\phi \mathrel{=\!\!|\!\vDash} \{\psi\{t/x\} \mid t : \sigma\} \mathrel{=\!\!|\!\vDash} \bigcup_{t:\sigma} GI(\psi\{t/x\}) = GI(\phi)$. $\square$

Theorem 5.4.16 (Herbrand's Theorem for NL). *If Γ is И∀-theory of NL then Γ is satisfiable if and only if it has a nominal Herbrand model. In addition, if Γ is unsatisfiable then there is a finite set of ground instances of formulas in Γ that is unsatisfiable.*

Proof. For the forward direction, if Γ is satisfiable, it has some model $\mathcal{M}$. Let $B = \{A \in B_{\mathcal{L}} \mid \mathcal{M} \vDash A\}$, the set of all basic formulas true in $\mathcal{M}$. Note that B must be equivariant since satisfiability in $\mathcal{M}$ is equivariant, so B is an Herbrand interpretation. Let $\mathcal{H}$ be the Herbrand model generated by B. Then by Proposition 5.4.13, if ϕ is nominal-universal and $\mathcal{M} \vDash \phi$ then $\mathcal{H} \vDash \phi$. In particular, for any $\phi \in \Gamma$, we have $\mathcal{M} \vDash \phi$ so $\mathcal{H} \vDash \phi$ since ϕ is И∀, and so $\mathcal{H} \vDash \Gamma$.

For the reverse direction and second part, if Γ is unsatisfiable, then $\Gamma \dashv\vDash GI(\Gamma)$, so the latter is unsatisfiable. Moreover, by compactness, $GI(\Gamma)$ has a finite unsatisfiable subset. □

We conclude with an important consequence of Herbrand's theorem which we will need later. Specifically, for sufficiently simple formulas such as closed atomic formulas and propositional combinations thereof, validity relative to $\mathcal{P}$ can be tested by considering only Herbrand models.

Corollary 5.4.17. *Let Γ be a closed И∀-theory and ϕ a closed quantifier-free formula. Then $\Gamma \vDash \phi$ if and only if for every Herbrand model $\mathcal{H} \vDash \Gamma$ we have $\mathcal{H} \vDash \phi$.*

Proof. For the forward direction, if $\Gamma \vDash \phi$ then all models of Γ, including Herbrand ones, model ϕ. For the reverse direction we prove the contrapositive. If $\Gamma \nvDash \phi$ then for some model $\mathcal{M} \vDash \Gamma$, $\mathcal{M} \vDash \neg\phi$. Hence, $\Gamma \cup \{\neg\phi\}$ is satisfiable; moreover since ϕ is quantifier free, $\neg\phi$ is И∀ so $\Gamma \cup \{\neg\phi\}$ is still a nominal-universal theory. Hence there must be an Herbrand model $\mathcal{H} \vDash \Gamma \cup \{\neg\phi\}$, hence $\mathcal{H} \nvDash \phi$. □

5.5 Notes

We have developed a semantics with respect to which nominal logic is complete. Pitts left the question of finding such a semantics open [108], and Gabbay proposed an infinitary rule for recovering completeness with respect to finitely supported models [40]. More recently, Gabbay [44] has investigated a more general approach to support, in which the set of names is uncountable and all supports are required to be at most countable. It is also possible to prove that nominal logic is complete with respect to such models as well; however, the presence of uncountable sets means that the Skolem-Löwenheim theorem fails, and such a model theory omits potentially interesting models with "countably infinite but small" supports.

The concept of support ideals is not new; the closely related structure of subgroup filters is used in Fraenkel-Mostowski set theory to prove the relative independence of variants of the Axiom of Choice and other principles; see Truss [124] and Felgner [35] for a discussion of this method.

Chapter 6

Nominal Logic Programming

Logic is the beginning of wisdom, not the end.

—*Spock*

The traditional approach to the semantics of logic programming, pioneered by van Emden, Kowalski and Apt [128, 8], is to define an *operational semantics* describing the possible program states and transitions performed by an idealized, nondeterministic interpreter for the language, define a *denotational* (or *declarative*) *semantics* identifying a program with a canonical model of the formulas comprising the program, and show that the two semantics agree. That is, any solution to a query found by the operational semantics is a correct answer, and if a query has an answer in the denotational semantics then this answer (or a generalization thereof) can be found by the operational semantics.

In this chapter, we study the semantics of nominal logic programming based on Horn clauses. In Section 6.1, we present an idealized, nondeterministic operational semantics for αProlog as a set of state-transition rules. Since αProlog programs involve freshness constraints, this semantics is based on that of *constraint logic programming* [61, 62]. In Section 6.2, we review and extend the results concerning Herbrand models of И$\forall$ theories in Section 5.4 and show that nominal Horn clause programs possess *least Herbrand models*, providing a van Emden-Kowalski style least-fixed-point semantics for αProlog programs. In Section 6.3, we prove that the operational and denotational semantics agree: if G is derivable from $\mathcal{P}$, then it is true in the least Herbrand model of $\mathcal{P}$ (soundness); conversely, if G is true in the least Herbrand model, then G is derivable in the operational semantics (completeness). Following Jaffar et al., we prove both *algebraic* and *logical* versions of the soundness and completeness results.

6.1 Operational Semantics

An operational semantics models the behavior of an abstract machine or interpreter for a language. For a logic programming language, the machine state is a set of subgoals remaining to be solved. The allowed transitions among states correspond to backchaining; transitions are labeled by substitutions. A query G is successful if there is a path from the state G to $\varnothing$; the answer to a query is the concatenation of the substitutions along the path.

In αProlog, the presence of freshness subproblems complicates this picture. We wish to allow answers to be qualified by (satisfiable) freshness subproblems, for example, to permit the answer $\{\mathsf{a} \# X, \mathsf{b} \# X\}$ to the query $(\mathsf{a}\ \mathsf{b}) \cdot X = X$. Constraint logic programming offers a suitable framework for dealing with freshness constraints; in the operational semantics of a CLP language, a state consists of a pair $\langle G \mid C\rangle$ consisting of the remaining subgoals G and a set of constraints C. From the CLP point of view, unification is just another constraint solving procedure and substitutions may be regarded as equational constraints in solved form. In fact, this insight leads to a considerably simpler presentation of the semantics, since the low-level details of constraint solving algorithms can be separated from the high-level issues of deduction, soundness, and completeness. Of course, unification and constraint solving algorithms must still be developed to obtain a practical implementation, but this is a separate issue, which we study in Chapter 7.

In αProlog, the constraints consist of equational constraints $t \approx u$, freshness constraints $t \# u$, and logical equivalence constraints $A \iff A'$. We abbreviate $A \iff A'$ as $A \sim A'$. In contrast to ordinary (constraint) logic programming, the constraint $p(\vec{t}) \sim p(\vec{u})$ is not equivalent to $\vec{t} \approx \vec{u}$, because of equivariance. For example, $p(\mathsf{a}) \sim (\mathsf{a}\ \mathsf{b}) \cdot p(\mathsf{a}) \sim p(\mathsf{b})$ but $\mathsf{a} \not\approx \mathsf{b}$.

Definition 6.1.1 (Constraints). *The* atomic constraints *c are $t \approx u$ and $t \# u$. A* compound constraint *C is a sequence $c_1, \ldots, c_n$ of atomic constraints, considered to indicate the conjunction of the listed constraints. Constraints are well-formed if the formulas comprising them are well-formed.*

The atomic constraints are interpreted as the corresponding nominal logic formulas, and compound constraints C are interpreted as conjunctions of the formulas in C.

Definition 6.1.2 (Goals and Programs). *A* goal *G is a conjunction of atomic formulas and constraints. A* nominal Horn clause *or* program clause *is a closed formula $\text{И}\vec{\mathsf{a}}.\forall\vec{X}.G \supset A$, often abbreviated $A :- G$, where A is a basic formula and G is a goal. A program $\mathcal{P}$ is a set of program clauses. Goals, program clauses, and programs are well-formed provided their component formulas are well-formed.*

Definition 6.1.3 (State). *A state is a pair $\Sigma : \langle G \mid C\rangle$, where G is a goal and C a constraint. States are well-formed provided G and C are simultaneously well-formed with respect to Σ.*

The transition rules in Table 6.1 express the operational semantics of αProlog programs. It is assumed in each rule that C is a set of constraints, $\mathcal{P}$ is a set of И$\forall$-closed program clauses, G is a sequence of atomic formulas, and all formulas

Table 6.1: Operational semantics rules for αProlog

	$G \longrightarrow G'$	provided
C	$\Sigma : \langle c, G \mid C\rangle \longrightarrow \Sigma : \langle G \mid C, c\rangle$	C, c consistent
S	$\Sigma : \langle G \mid C\rangle \longrightarrow \Sigma;\Sigma' : \langle G \mid C'\rangle$	$\Sigma;\Sigma' : C' \vdash C$
B	$\Sigma : \langle A, G \mid C\rangle \longrightarrow \Sigma\#\vec{\mathsf{a}}, \vec{X} : \langle A' \sim A, G', G \mid C\rangle$	$\text{И}\vec{\mathsf{a}}.\forall\vec{X}.G' \supset A' \in \mathcal{P}$

occurring in $C, G, \mathcal{P}$, are well-formed. A query G is successful if there is a path from $\langle G \mid \varnothing\rangle$ to a satisfiable constraint set $\langle \varnothing \mid C\rangle$, and the resulting answer is C.

The simplification rule $\longrightarrow_S$ deserves some explanation. It is used to simplify constraints. For example, if $C = f(X, \langle\mathsf{a}\rangle X) \approx f(Y, \langle\mathsf{b}\rangle\mathsf{b})$, then C can be replaced by the simpler constraint $C' = X \approx \mathsf{a}, Y \approx \mathsf{a}$, since C' corresponds to a unifier of C. Put another way, $C' \vDash C$, so the simplification rule can be used to transition from C to C'. Also, as we shall see in Chapter 7, some constraint solving steps introduce additional name-constants, so we permit the context to grow.

Note that in the simplification example just given, $C \vDash C'$ as well, so we could "simplify" by replacing the unifier C' with the original problem C. There is nothing logically wrong with this. In this chapter we wish to abstract away from the details of how constraints are solved, and this includes such matters as the definition of *solved forms* which are considered to be simplified. These matters are addressed in Chapter 7, and the S-rule is a high-level abstraction for all of the constraint simplification algorithms developed there.

Remark 6.1.4 (Backchaining and equivariance). Equivariance is the biggest complication in the operational semantics of αProlog. In Table 6.1, the C and S rules are standard from constraint logic programming. However, the B rule is nonstandard. In ordinary CLP, the backchaining rule is as follows:

$$\Sigma : \langle A, G \mid C\rangle \longrightarrow_{B_\approx} \Sigma, \vec{X} : \langle \vec{t} \approx \vec{u}, G', G \mid C\rangle \quad (\forall\vec{X}.G' \supset A' \in \mathcal{P})$$

where $A = p(\vec{t}), A' = p(\vec{u})$. That is, in ordinary (constraint) logic programming, term equality is sufficient to decide equivalence of atomic formulas, and syntactic unification is used in backchaining. In nominal logic programming, it might seem natural to extend this definition, using nominal equality and unification instead of first-order equality and unification:

$$\Sigma : \langle A, G \mid C\rangle \longrightarrow_{B_\approx} \Sigma\#\vec{\mathsf{a}}, \vec{X} : \langle \vec{t} \approx \vec{u}, G', G \mid C\rangle \quad (\text{И}\vec{\mathsf{a}}.\forall\vec{X}.G' \supset A' \in \mathcal{P})$$

This approach is sound: all the answers derivable by backchaining based on nominal equality are correct with respect to the denotational semantics we present in the next section. However, it is *not* complete, for there are programs and queries that have answers that cannot be derived using $B_\approx$, because the names generated during a backchaining step are required to be fresh.

For example, given the program

$$\text{И}\mathsf{a}.p(\mathsf{a}).$$

the query $p(\mathsf{b})$ has no answer, since the following partial derivation gets stuck and there are no other possible derivations:

$$\mathsf{b} : \langle p(\mathsf{b}) \mid \varnothing \rangle \longrightarrow_{B_\approx} \mathsf{b}\#\mathsf{a} : \langle p(\mathsf{b}) \approx p(\mathsf{a}) \mid \varnothing \rangle$$

There is an alternative backchaining rule based on equality that *is* complete:

$$\Sigma : \langle A, G \mid C \rangle \longrightarrow_{B'_\approx} \Sigma\#\vec{\mathsf{b}}, \vec{X} : \langle A' \approx A, G', G \mid C \rangle \quad (\text{И}\vec{\mathsf{a}}.\forall \vec{X}.G' \supset A' \in \mathcal{P})$$

where $\vec{\mathsf{b}} \subseteq \vec{\mathsf{a}}$. This rule relaxes the requirement that all the И-quantified names a be chosen fresh. Instead, some "stale" names may be chosen.

Using $B'_\approx$, we can derive

$$\mathsf{b} : \langle p(\mathsf{b}) \mid \varnothing \rangle \longrightarrow_{B'_\approx} \mathsf{b} : \langle p(\mathsf{b}) \approx p(\mathsf{b}) \mid \varnothing \rangle$$

by instantiating the program clause using the stale name b.

This rule is, however, highly nondeterministic. It provides no guidance as to how the names in a program clause are to be instantiated. In fact, this is a hard problem, and it is equivalent to the problem of solving constraints of the form $A \sim A'$.

We prefer the approach taken in rule B for two reasons. First, it reduces the nondeterminism in the operational semantics to an acceptable level. While the underlying hard search problem is still there, it is kept separate from the high-level logical behavior of the program. Second, it corresponds closely to an operational interpretation of the respective left-rules for И, $\forall$, and $\supset$, providing a direct intuitive justification of backchaining in terms of our proof system for nominal logic.

Now we show an important, but straightforward, type soundness property for monomorphic αProlog programs. Although the implementation of αProlog provides polymorphic typing, we have not yet investigated the combination of polymorphism and nominal logic. Investigation of type soundness for polymorphic programs is left for future work.

Theorem 6.1.5 (Type soundness). *If $\mathcal{P}$ is a well-formed program and S is a well-formed state and $\Sigma : S \longrightarrow^* \Sigma' : S'$ using the transitions of Table 6.1 then $\Sigma' : S'$ is well-formed.*

Proof. The proof is routine based on the assumption that G and $\mathcal{P}$ are well-formed, and that in the simplification rule, implicitly C' must be well-formed. □

6.1.1 Examples

Example 6.1.6 (Typechecking). We first demonstrate how to solve a query $tc([], lam(\langle \mathsf{a}\rangle lam(\langle \mathsf{a}\rangle var(\mathsf{a}))), T)$. The context Σ and many intermediate steps are left out for conciseness.

$$\begin{array}{ll}
 & \langle tc([], lam(\langle \mathsf{a}\rangle lam(\langle \mathsf{a}\rangle var(\mathsf{a}))), T) \mid \varnothing\rangle \\
\longrightarrow_B & \langle \mathsf{x}_1 \# [], tc([(\mathsf{x}_1, T_1)|G_1], E_1, U_1) \mid \\
 & tc([], lam(\langle \mathsf{a}\rangle lam(\langle \mathsf{a}\rangle var(\mathsf{a}))), T) \sim tc(G_1, lam(\langle \mathsf{x}_1\rangle E_1), arr(T_1, U_1))\rangle \\
\longrightarrow_S & \langle tc([(\mathsf{x}_1, T_1)|G_1], E_1, U_1) \mid \\
 & T \approx arr(T_1, U_1), G_1 \approx [], E_1 \approx lam(\langle \mathsf{x}_1\rangle var(\mathsf{x}_1))\rangle \\
\longrightarrow^* & \langle mem((X_3, T_3), G_3) \mid tc([(\mathsf{x}_2, T_2), G_2], E_1, U_2) \sim tc(G_3, var(X_3), T_3) \\
 & \mathsf{x}_2 \# T_1, T \approx arr(T_1, arr(T_2, U_2)), G_2 \approx [(\mathsf{x}_1, T_1)], E_2 \approx var(\mathsf{x}_2)\rangle \\
\longrightarrow^* & \langle \varnothing \mid \mathsf{x}_2 \# T_1, T \approx arr(T_1, arr(T_2, T_2))\rangle
\end{array}$$

Thus the answer for T is $T \approx arr(T_1, arr(T_2, T_2))$ as desired. Note that $\mathsf{x}_2 \# T_1$ can be simplified away because at this stage, the context Σ is of the form $\mathsf{a}, T\#\mathsf{x}_1, G_1, T_1, E_1, U_1\#\mathsf{x}_2, \ldots$ so $\Sigma : C \vdash \mathsf{x}_2 \# T_1 \wedge C$ follows by the $\Sigma\#$ rule (writing $\phi = T \approx arr(T_1, arr(T_2, T_2))$). There are no essentially different answers to the initial query. In the previous example, all of the equivariance constraints were trivial, that is, the equivariance axioms are not needed to solve the atomic $\sim$-constraints.

Example 6.1.7 (Equivariance). The α-inequivalence predicate *neq* is a typical program for which equivariance is needed. Consider the following execution:

$$\begin{array}{ll}
 & neq(lam(\langle \mathsf{x}\rangle var(\mathsf{y})), lam(\langle \mathsf{z}\rangle var(\mathsf{z}))) \\
\longrightarrow_B & \langle neq(lam(\langle \mathsf{x}\rangle var(\mathsf{y})), lam(\langle \mathsf{z}\rangle var(\mathsf{z}))) \sim neq(lam(\langle \mathsf{x}_1\rangle E), lam(\langle \mathsf{x}_1\rangle E'), \\
 & neq(E, E') \mid \varnothing\rangle \\
\longrightarrow_S & \langle neq(E, E') \mid E \approx var(\mathsf{y}), E' \approx var(\mathsf{x}_1)\rangle \\
\longrightarrow_B & \langle neq(var(\mathsf{y}), var(\mathsf{x}_1)) \sim neq(var(\mathsf{x}_2), var(\mathsf{y}_2)) \mid \cdots\rangle \\
\longrightarrow_S & E \approx var(\mathsf{y}), E' \approx var(\mathsf{x}_1)
\end{array}$$

where $neq(var(\mathsf{y}), var(\mathsf{x}_1)) \sim neq(var(\mathsf{x}_2), var(\mathsf{y}_2))$ requires equivariance so that the permutation $(\mathsf{y}\ \mathsf{x}_2)(\mathsf{x}_1\ \mathsf{y}_2)$ can be applied.

6.2 Denotational Semantics

A denotational semantics gives mathematical meaning to expressions and programs in a programming language. Basic datatypes such as integers and strings are associated with mathematical integers and lists of characters; procedures are associated with functions; and basic formulas are associated with relations.

The standard approach to denotational semantics for logic programming is to view a program as a theory (in an appropriate logic) and associate it with some canonical model (or a class of models). In first-order logic programming, the natural domain in which to interpret terms and program clauses is the first-order Herbrand universe consisting of terms and Herbrand models. In fact, first-order Horn clause programs possess unique *least Herbrand models*. Moreover, such models can also be obtained by calculating the least fixed point of an appropriate continuous operator. This property is essential for proving completeness.

This section relies heavily on definitions and concepts from lattice theory, which are reviewed in Section 3.4. We now 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 one-step deduction operator, following the development given by Lloyd [69] and Jaffar et al. [62].

6.2.1 Least Herbrand Models

It is a well-known fact that least Herbrand models exist for Horn clause theories in first-order logic. Building on the nominal version of Herbrand's Theorem, we extend this result to nominal logic.

Proposition 6.2.1. *Let $\mathcal{P}$ be a program and $\mathcal{M}$ a nonempty set of Herbrand models of $\mathcal{P}$. Then $\bigcap \mathcal{M}$ is also a Herbrand model of $\mathcal{P}$.*

Proof. Obviously, the sort and term interpretation of $\bigcap \mathcal{M}$ is the Herbrand interpretation. We therefore need to show that $\bigcap \mathcal{M} \vDash \phi$ for each $\phi \in \mathcal{P}$. For this we rely heavily on Corollary 5.4.11, which states that a nominal-universal formula $\mathsf{N}\vec{\mathsf{a}}.\forall \vec{X}.\phi$ is valid in a Herbrand model $\mathcal{H}$ just in case $\mathcal{H} \vDash \theta(\phi)$ for every valuation θ. The proof is otherwise standard, but we give the details for completeness.

Let ϕ be a rule A :− G. Then for every $\mathcal{H} \in \mathcal{M}$, $\mathcal{H} \vDash \phi$, so we know that whenever $\mathcal{H} \vDash \theta(G)$, it follows that $\mathcal{H} \vDash \theta(A)$. Assume that $\bigcap \mathcal{M} \vDash \theta(G)$ for some θ. Let $\mathcal{H} \in \mathcal{M}$ be given. Clearly, $\mathcal{H} \vDash \theta(G)$ so $\mathcal{H} \vDash \theta(A)$. Since $\mathcal{H}$ was arbitrary, we have $\theta(A) \in \bigcap \mathcal{M}$, i.e. $\bigcap \mathcal{M} \vDash \theta(A)$. Therefore, $\bigcap \mathcal{M} \vDash A :- G$. Since $\bigcap \mathcal{M} \vDash \phi$ for any $\phi \in \mathcal{P}$, we have $\bigcap \mathcal{M} \vDash \mathcal{P}$, so $\bigcap \mathcal{M}$ is a Herbrand model of $\mathcal{P}$. □

An immediate consequence is that a $\subseteq$-least Herbrand model

$$\mathcal{H}_{\mathcal{P}} = \bigcap \{\mathcal{H} \mid \mathcal{H} \vDash \mathcal{P}\}$$

exists for any nominal Horn theory $\mathcal{P}$. Moreover, $\mathcal{H}_{\mathcal{P}}$ consists of all ground atoms provable from $\mathcal{P}$, as we now show.

Theorem 6.2.2. *Let $\mathcal{P}$ be a set of program clauses. Then $\mathcal{H}_{\mathcal{P}} = \{A \in B_{\mathcal{L}} \mid \mathcal{P} \vDash A\}$.*

Proof. If $A \in \mathcal{H}_{\mathcal{P}}$, then A is valid in every Herbrand model of $\mathcal{P}$, so by Corollary 5.4.17, A is valid in every model of $\mathcal{P}$. Conversely, if $\mathcal{P} \vDash A$ then since $\mathcal{H}_{\mathcal{P}} \vDash \mathcal{P}$ we have $\mathcal{H}_{\mathcal{P}} \vDash A$; thus $A \in \mathcal{H}_{\mathcal{P}}$. □

6.2.2 Fixed Point Semantics

Note that the set of all Herbrand interpretations $\{B \subset \mathcal{P}(B_{\mathcal{L}}) \mid supp(S) = \varnothing\}$ forms a complete nominal lattice with a top element $B_{\mathcal{L}}$ and bottom element $\varnothing$. The join and meet operations are $\cup$ and $\cap$, respectively.

Definition 6.2.3. *Let S be a Herbrand interpretation and $\mathcal{P}$ a set of program clauses. Define the transformation $\tau_{\mathcal{P}} : \mathcal{P}(B_{\mathcal{L}}) \to \mathcal{P}(B_{\mathcal{L}})$ by*

$$\tau_{\mathcal{P}}(S) = \{\theta(A) \mid (A :\!-\, G) \in \mathcal{P}, S \vDash \theta(G)\}$$

where θ is a ground valuation.

Proposition 6.2.4. *$\tau_{\mathcal{P}}$ is monotone, equivariant, and continuous.*

Proof. It is easy to see that $\tau_{\mathcal{P}}$ is monotone.

For equivariance, let compatible names a, b be given. Then

$$\begin{array}{rcl}
(a\ b) \cdot \tau_{\mathcal{P}}(S) & = & (a\ b) \cdot \{\theta(A) \mid (A :\!-\, G) \in \mathcal{P}, S \vDash \theta(G)\} \\
& = & \{(a\ b) \cdot \theta(A) \mid (A :\!-\, G) \in \mathcal{P}, S \vDash \theta(G)\} \\
& = & \{\theta((a\ b) \cdot A) \mid (A :\!-\, G) \in \mathcal{P}, S \vDash \theta(G)\} \\
& = & \{\theta((a\ b) \cdot A) \mid ((a\ b) \cdot A :\!-\, (a\ b) \cdot G) \in \mathcal{P}, \\
& & \quad (a\ b) \cdot S \vDash \theta((a\ b) \cdot G)\} \\
& = & \{\theta(A) \mid (A :\!-\, G) \in \mathcal{P}, (a\ b) \cdot S \vDash \theta(G)\} \\
& = & \tau_{\mathcal{P}}((a\ b) \cdot S)
\end{array}$$

For continuity, let $\vec{S}$ be an ω-chain of subsets of $B_{\mathcal{P}}$. We need to show that

$$\tau_{\mathcal{P}}(\bigcup_i S_i) = \bigcup_i \tau_{\mathcal{P}}(S_i)\ .$$

For the $\subseteq$ direction, suppose that $A \in \tau_{\mathcal{P}}(\bigcup_i S_i)$. Then $A = \theta(A')$ and

$$\theta(G_1), \ldots, \theta(G_n) \in \bigcup_i S_i$$

for some clause $A' :\!-\, G' \in \mathcal{P}$. Since there are finitely many G_i, there is a fixed S_j such that $\theta(G_1), \ldots, \theta(G_n) \in S_j$. Hence $A \in \tau(S_j) \subseteq \bigcup_i \tau_{\mathcal{P}}(S_i)$.

For the $\supseteq$ direction, suppose that $A \in \bigcup_i \tau_{\mathcal{P}}(S_i)$. Then for some j, $A \in \tau_{\mathcal{P}}(S_j)$, hence $A = \theta(A')$ and $\theta(G_1), \ldots, \theta(G_n) \in S_j \subseteq \bigcup_i S_i$. Consequently, $A \in \tau_{\mathcal{P}}(\bigcup_i S_i)$. □

Theorem 6.2.5. *$\mathcal{H}_{\mathcal{P}} = lfp(\tau_{\mathcal{P}}) = \tau_{\mathcal{P}}^{\omega}(\varnothing)$.*

Proof. Clearly $\tau_{\mathcal{P}}^{\omega}(\varnothing) = lfp(\tau_{\mathcal{P}})$ by Theorem 3.4.4. We show that $\mathcal{H}_{\mathcal{P}} \subseteq \tau_{\mathcal{P}}^{\omega}(\varnothing)$ and $lfp(\tau_{\mathcal{P}}) \subseteq \mathcal{H}_{\mathcal{P}}$.

For $\mathcal{H}_{\mathcal{P}} \subseteq \tau_{\mathcal{P}}^{\omega}(\varnothing)$, we show that $\mathcal{M} = \tau_{\mathcal{P}}^{\omega}(\varnothing)$ is a model of $\mathcal{P}$. To see that $\mathcal{M}$ is a model of $\mathcal{P}$, we consider each clause A :− G of $\mathcal{P}$. We wish to show that $\mathcal{M} \vDash A$:− G, that is, for any ground valuation θ, $\mathcal{M} \vDash \theta(G) \supset \theta(A)$. Let θ be a substitution and assume $\mathcal{M} \vDash \theta(G)$. By the definition of $\tau_{\mathcal{P}}$ we have $\theta(A) \in \tau_{\mathcal{P}}(\mathcal{M})$, but since $\mathcal{M}$ is a fixed point of $\tau_{\mathcal{P}}$ we have $\tau_{\mathcal{P}}(\mathcal{M}) = \mathcal{M}$. Hence $\theta(A) \in \mathcal{M}$, so A :− G is valid in $\mathcal{M}$, and $\mathcal{M} \vDash \mathcal{P}$, as desired. Because $\mathcal{H}_{\mathcal{P}}$ is the least model of $\mathcal{P}$, $\mathcal{H}_{\mathcal{P}} \subseteq \tau_{\mathcal{P}}^{\omega}(\varnothing)$.

For $lfp(\tau_{\mathcal{P}}) \subseteq \mathcal{H}_{\mathcal{P}}$, we show that $\mathcal{H}_{\mathcal{P}}$ is a pre-fixed point of $\tau_{\mathcal{P}}$, that is, $\tau_{\mathcal{P}}(\mathcal{H}_{\mathcal{P}}) \subseteq \mathcal{H}_{\mathcal{P}}$. Suppose $A \in \tau_{\mathcal{P}}(\mathcal{H}_{\mathcal{P}})$. Then for some A' :− $G' \in \mathcal{P}$, and some substitution θ, we have $A = \theta(A')$ and $\mathcal{H}_{\mathcal{P}} \vDash \theta(G')$. Since $\mathcal{H}_{\mathcal{P}} \vDash \mathcal{P}$, and $\mathcal{H}_{\mathcal{P}} \vDash \theta(G')$, it follows that $\mathcal{H}_{\mathcal{P}} \vDash \theta(A')$, so $\mathcal{H}_{\mathcal{P}} \vDash A$. Hence $A \in \mathcal{H}_{\mathcal{P}}$, as desired, so $\mathcal{H}_{\mathcal{P}}$ is a pre-fixed point of $\tau_{\mathcal{P}}$. But $\tau_{\mathcal{P}}^{\omega}(\varnothing)$ is the least fixed point of $\tau_{\mathcal{P}}$, and by the Knaster-Tarski theorem, the least fixed point is the least pre-fixed point, so the desired result is immediate. □

We have thus extended the standard denotational semantics for first-order logic programs to the nominal case.

6.2.3 Examples

We consider some simple example programs and their least models.

Consider the program

$$p(\mathsf{a}, X) :- \mathsf{a} \approx X$$

The corresponding transformation τ is

$$\tau_{\mathcal{P}}(S) = \{p(\mathsf{a}, v) \mid S \vDash \mathsf{a} \approx v\}$$

This operator reaches a fixed point after one step, namely

$$\mathcal{H}_{\mathcal{P}} = \{p(\mathsf{a}, \mathsf{a}) \mid \mathsf{a} \in \nu\}$$

Consider the program

$$\begin{array}{lll} fv(var(\mathsf{x}), [\mathsf{x}]). & & \\ fv(app(E, E'), L) & :- & fv(E, L_1), fv(E', L_2), union(L_1, L_2, L). \\ fv(lam(\langle \mathsf{x} \rangle E), L) & :- & fv(E, L'), remove(\mathsf{x}, L', L). \end{array}$$

where *union* merges two lists without introducing repeated elements. The least model for fv looks something like this:

$$\begin{array}{ll} & \{fv(var(\mathsf{x}), [\mathsf{x}]), \ldots\} \\ \cup & \{fv(app(var(\mathsf{x}), var(\mathsf{y})), [\mathsf{x}, \mathsf{y}]), \ldots\} \\ \cup & \{fv(lam(\langle \mathsf{x} \rangle app(var(\mathsf{x}), var(\mathsf{y}))), [\mathsf{y}]), \ldots\} \end{array}$$

6.3 Soundness and Completeness

Soundness and completeness for the operational semantics relative to the denotational semantics is one of the most important properties to establish for any logic programming language. For first-order Horn clause programs, a ground formula is true in the least Herbrand model $\mathcal{H}_{\mathcal{P}}$ iff it has a successful derivation in the operational semantics. These soundness and completeness results are crucial for establishing that logic programs are both algorithmically and logically sensible.

Soundness and completeness properties for constraint logic programming come in two varieties: *algebraic*, and *logical*. Algebraic soundness and completeness mean that any concrete answer (ground valuation) solving a query can be derived using the operational semantics and vice versa. Logical soundness and completeness mean that any query is logically equivalent to a disjunction of constraints, and is typically harder to establish. This terminology is due to Jaffar et al. [62].

In this section we first show algebraic soundness, then algebraic completeness, and finally logical soundness and completeness.

6.3.1 Algebraic Soundness

Theorem 6.3.1 (Algebraic Soundness). *If* $\Sigma : \langle G \mid \varnothing\rangle \longrightarrow^* \Sigma;\Sigma' : \langle \varnothing \mid C\rangle$ *and* $\mathcal{M} \vDash \mathcal{P}$ *then* $\Sigma;\Sigma' : C \vDash^{\mathcal{M}} G$.

Proof. Let $\mathcal{M} \vDash \mathcal{P}$ be given. First we show that if $\Sigma : \langle G \mid C\rangle \longrightarrow \Sigma;\Sigma' : \langle G' \mid C'\rangle$ then $\Sigma;\Sigma' : G', C' \vDash G, C$. We consider the possible cases.

- If the transition was of the form

$$\Sigma : \langle c, G \mid C\rangle \longrightarrow_C \Sigma : \langle G \mid C, c\rangle$$

  then it is immediate that $\Sigma : c, G, C \vDash G, C, c$.

- If the transition was of the form

$$\Sigma : \langle G \mid C\rangle \longrightarrow_S \Sigma;\Sigma' : \langle G \mid C'\rangle$$

  where $\Sigma;\Sigma' : C' \Rightarrow C$ then clearly, by the soundness of NL, we have $\Sigma;\Sigma' : C' \vDash C$ so we also have $\Sigma;\Sigma' : G, C' \vDash G, C$.

- Finally, if the transition was of the form

$$\Sigma : \langle A, G \mid C\rangle \longrightarrow_B \Sigma;\Sigma' : \langle A' \sim A, G', G \mid C\rangle$$

  where $\Sigma' = \#\vec{\mathsf{a}}, \vec{X}$, with $\text{И}\vec{\mathsf{a}}.\forall\vec{X}.G' \supset A' \in \mathcal{P}$, then we need to show $\Sigma;\Sigma' : A' \sim A, G', G, C \vDash A, G, C$. Let $\rho : \Sigma;\Sigma'$ be an $\mathcal{M}$-valuation such that $\Sigma;\Sigma' : \rho \vDash A' \sim A, G', G, C$. We need to show $\Sigma;\Sigma' : \rho \vDash A, G, C$. Obviously $\Sigma;\Sigma' : \rho \vDash G, C$ by assumption, so the only difficult part is showing $\Sigma;\Sigma' : \rho \vDash A$. However, since $\text{И}\vec{\mathsf{a}}.\forall\vec{X}.G' \supset A' \in \mathcal{P}$, and $\mathcal{M} \vDash \mathcal{P}$, we must have

$\Sigma; \Sigma' : \rho \vDash G' \supset A'$, so combined with the fact that $\Sigma; \Sigma' : \rho \vDash G'$, we have $\Sigma; \Sigma' : \rho \vDash A'$. Since $\Sigma; \Sigma' : \rho \vDash A' \sim A$, we also have $\Sigma; \Sigma' : \rho \vDash A$. Since ρ was arbitrary, we have $\Sigma; \Sigma' : A' \sim A, G', G, C \vDash A, G, C$, as desired.

It follows immediately that if $\Sigma : \langle G \mid C' \rangle \longrightarrow^* \Sigma' : \langle G' \mid C \rangle$ then $\Sigma' : G', C \vDash G, C'$; the desired result follows if $C' = \varnothing$ and $G' = \varnothing$. □

6.3.2 Algebraic Completeness

We now consider the question of (algebraic) completeness: if a goal is satisfiable, then it is operationally reducible to a satisfiable constraint. Since $\mathcal{P}$ is a nominal-universal Horn theory, any satisfiable formula is satisfiable in $\mathcal{H}_\mathcal{P}$. The idea of the proof is to exploit the fact that $\mathcal{H}_\mathcal{P}$ is constructed by a fixpoint operator.

Some auxiliary definitions and notation involving valuations are now given.

Definition 6.3.2. *We say that ground valuations θ and θ' are* disjoint *if their domains are disjoint; the* sum *of two disjoint valuations θ, θ' is $\theta + \theta'$, defined by*

$$(\theta + \theta')(X) = \begin{cases} \theta(X) & X \in dom(\theta) \\ \theta'(X) & X \in dom(\theta') \end{cases}$$

We say that θ' extends *θ (written $\theta' \geq \theta$) if there exists a θ'' such that $\theta' = \theta + \theta''$.*

It is easy to verify the following properties of valuations, sums, and extensions:

Lemma 6.3.3. *If θ, θ' are disjoint valuations then $\theta + \theta'$ is a valuation.*
If $\Sigma : \mathcal{M}, \theta \vDash \phi$, $\theta' : \Sigma'$, $\Sigma' \geq \Sigma$, and $\theta' \geq \theta$ then $\Sigma' : \mathcal{M}, \theta' \vDash \phi$.

Theorem 6.3.4 (Algebraic Completeness). *Assume $\Sigma : \mathcal{H}_\mathcal{P}, \theta \vDash G$. Then there exists a constraint C, context $\Sigma' \geq \Sigma$, and valuation $\theta' \geq \theta$ such that $\Sigma : G \longrightarrow^* \Sigma' : C$ and $\Sigma' : \theta' \vDash C$.*

Proof. Since $\mathcal{H}_\mathcal{P}, \theta \vDash G$ and $\mathcal{H}_\mathcal{P} = \tau_\mathcal{P}^\omega$, there must be a finite iteration $\tau_i = \tau_\mathcal{P}^i(\varnothing)$ of $\tau_\mathcal{P}$ such that $\tau_i, \theta \vDash G$. Therefore, we prove by induction on i that if $\tau_i, \theta \vDash G$ then there exists Σ', θ', C such that $\Sigma : G \longrightarrow^* \Sigma' : C$, $\Sigma' \geq \Sigma$, $\theta' \geq \theta$, and $\Sigma' : \theta' \vDash C$.

If $\tau_0 = \varnothing$, then $\vDash G$, so G must consist entirely of constraints, and we have

$$\Sigma : \langle G \mid \varnothing \rangle \longrightarrow_C \Sigma : \langle \varnothing \mid G \rangle .$$

The desired constraint is G itself and the desired valuation is θ.

For the induction step, assume the induction hypothesis holds for $i = n$. Suppose $\Sigma : \tau_{n+1}, \theta \vDash G$. We proceed by induction on the number of goals in G. If $|G| = 0$, then G is empty, so $\Sigma : G \longrightarrow^* \Sigma : \top$. Otherwise, $|G| = k + 1$ for some k such that the (inner) induction hypothesis holds for k. Let $G = G', A$; then $|G'| = k$ and $\Sigma : \tau_{n+1}, \theta \vDash G'$ so by the inner induction hypothesis applied to G', choose C', $\Sigma' \geq \Sigma$, and θ' such that $\Sigma : G' \longrightarrow^* \Sigma' : C'$ and $\Sigma' : \theta' \vDash C'$. Now, concerning A, there are two cases: either $\Sigma' : \tau_n, \theta \vDash A$ or $\Sigma' : \tau_n, \theta \nvDash A$. In the first case, clearly $\Sigma' : \tau_n, \theta' \vDash A$ so by the outer induction hypothesis there exists a C'' and $\theta'' \geq \theta'$ such that $A \longrightarrow^* C''$ and $\theta'' \vDash C''$, thus

$$\begin{array}{rcl} \Sigma : \langle G', A \mid \varnothing \rangle & \longrightarrow^* & \Sigma' : \langle C', A \mid \varnothing \rangle \\ & \longrightarrow_C & \Sigma' : \langle A \mid C' \rangle \\ & \longrightarrow^* & \Sigma'' : \langle C'' \mid C' \rangle \\ & \longrightarrow_C & \Sigma'' : \langle \varnothing \mid C'', C' \rangle \end{array}$$

so the desired C is C'', C', for which it is easily verified that $\Sigma'' : \theta'' \vDash C'', C'$.

Otherwise, $\Sigma : \tau_{n+1}, \theta \vDash A$ but $\Sigma : \tau_n, \theta \not\vDash A$. So it must be that $\theta(A) \in \tau(\tau_n) - \tau_n$, in other words, that there exists a θ_0 and $(\text{И}\vec{\mathsf{a}}.\forall \vec{X}.G_0 \supset A_0) \in \mathcal{P}$ such that $\vDash \theta_0(A_0) \sim \theta(A)$, $\#\vec{\mathsf{a}}, \vec{X} : \tau_n, \theta_0 \vDash G$, and $FV(A_0, G_0) \cap FV(A, G') = \varnothing$. Since the free variables of A and A_0, G_0 are distinct, θ' and θ_0 are disjoint; we define $\theta'' = \theta' + \theta_0$. Note that $\vDash \theta_0(A_0) \sim \theta(A)$ implies $\Sigma'' : \theta'' \vDash A_0 \sim A$ where $\Sigma'' = \Sigma' \# \vec{\mathsf{a}}, \vec{X}$. Then $\Sigma'' : \tau_n, \theta'' \vDash G_0$, so by the outer induction hypothesis on G_0, there exists C_0, Σ''', and θ''' such that $\Sigma'' : G_0 \longrightarrow^* \Sigma''' : C_0$ and $\Sigma''' : \theta''' \vDash C_0$. Now certainly $\Sigma''' : \theta''' \vDash C_0, A_0 \sim A, C'$ since Σ''' and θ''' extend all the contexts and valuations involved. Moreover,

$$\begin{array}{rcl} \Sigma : \langle G', A \mid \varnothing \rangle & \longrightarrow^* & \Sigma' : \langle C', A \mid \varnothing \rangle \\ & \longrightarrow_C & \Sigma' : \langle A \mid C' \rangle \\ & \longrightarrow_B & \Sigma'' : \langle A_0 \sim A, G_0 \mid C' \rangle \\ & \longrightarrow_C & \Sigma'' : \langle G_0 \mid A_0 \sim A, C' \rangle \\ & \longrightarrow^* & \Sigma''' : \langle C_0 \mid A_0 \sim A, C' \rangle \\ & \longrightarrow_C & \Sigma''' : \langle \varnothing \mid C_0, A_0 \sim A, C' \rangle \end{array}$$

and we can construct the desired context, constraint, and valuation. This exhausts all cases and completes the proof. □

Note that nowhere was the simplification rule $\longrightarrow_S$ needed. Hence the operational semantics consisting only of $\longrightarrow_B, \longrightarrow_C$ is complete. The simplification rule is only needed to reduce complicated constraints to human-readable forms.

6.3.3 Logical Soundness and Completeness

So far we have shown that completeness holds in the weak sense that a valid solution can be derived in the operational semantics. In this section we consider *logical completeness*, that is, that "the answers returned by the operational semantics cover all of the constraints which imply the goal [62]".

The usual way to prove this is to show that G is logically equivalent to the disjunction of *all* its derivable solutions (possibly infinitely many), and use compactness to show that there must be a finite subset of the disjunction. However, the presence of structured contexts in nominal logic means that we must be careful when dealing with potentially infinite sets of formulas since they may mention infinitely many variables and name-constants. One way to deal with this would be to use infinite contexts, as in Chapter 5; however, this approach is somewhat cumbersome for this situation. Instead, we will use auxiliary notation to represent the existential closure of a constraint with respect to a context.

Definition 6.3.5 (Nominal-Existential Closure). *Given a context Σ and constraint C, we write $\exists\Sigma[C]$ for the* nominal-existential closure *of C with respect to Σ, or the result of quantifying the name-constants of Σ using И and variables of Σ using $\exists$. This can be computed as follows:*

$$\begin{array}{rcl} \exists\cdot[C] &=& C \\ \exists\Sigma,x[C] &=& \exists\Sigma[\exists x.C] \\ \exists\Sigma\#\mathsf{a}[C] &=& \exists\Sigma[\text{И}\mathsf{a}.C] \end{array}$$

For example, if some context and goal, such as $\mathsf{a},T : tc([],lam(\langle\mathsf{a}\rangle var(\mathsf{a})),T)$, reduces to a constraint C in an extended context, such as $\mathsf{a},T,\mathsf{b},U : T\approx f(\langle\mathsf{b}\rangle U)$, then the И$\exists$-closure relative to the context U is is $\text{И}\mathsf{b}.\exists U[T\approx f(\langle\mathsf{b}\rangle U)] = \exists U.T \approx f(\langle\mathsf{b}\rangle U)$. It is not difficult to show that

Lemma 6.3.6. $\Sigma : \theta \vDash \exists\Sigma'[C]$ *if and only if there exists a $\theta' : \Sigma'$ such that $\Sigma;\Sigma' : \theta+\theta' \vDash C$.*

This, combined with the Algebraic Soundness Theorem above, gives a sharpened result that shows that if a constraint is derivable, then its closure under the context extension involved in the derivation is also a logical answer to the goal.

Corollary 6.3.7 (Logical Soundness). *If $\Sigma : G \longrightarrow^* \Sigma;\Sigma' : C$ then $\Sigma : \mathcal{P},\exists\Sigma'[C] \vDash G$.*

We now show that a goal is logically equivalent to the closures of all the constraints to which it reduces.

Proposition 6.3.8. *For given G and Σ, let $\mathcal{C} = \{\exists\Sigma'[C] \mid \Sigma : G \longrightarrow^* \Sigma;\Sigma' : C\}$. Then $\Sigma : \mathcal{H}_\mathcal{P} \vDash G \iff \bigvee\mathcal{C}$.*

Proof. For the forward direction, let $\mathcal{H}_\mathcal{P},\theta \vDash G$. By Theorem 6.3.4, there exists a Σ', θ', and C such that $\Sigma;\Sigma' : \theta+\theta' \vDash C$ and $\Sigma : G \longrightarrow^* \Sigma;\Sigma' : C$; moreover, $\Sigma : \theta \vDash \exists\Sigma'[C]$ by Lemma 6.3.6. Hence, $\Sigma : \theta \vDash \bigvee\mathcal{C}$, so $\mathcal{H}_\mathcal{P},\theta \vDash G \supset \bigvee\mathcal{C}$, so $\mathcal{H}_\mathcal{P} \vDash G \supset \bigvee\mathcal{C}$ since θ was arbitrary.

For the reverse direction, assume $\Sigma : \mathcal{H}_\mathcal{P},\theta \vDash \bigvee\mathcal{C}$. Then for some $\exists\Sigma'[C'] \in \mathcal{C}$, we have $\Sigma : \mathcal{H}_\mathcal{P},\theta \vDash \exists\Sigma'[C]$. By the definition of $\mathcal{C}$, we have $\Sigma : G \longrightarrow^* \Sigma;\Sigma' : C$. Therefore, by Corollary 6.3.7, $\Sigma : \mathcal{P},\exists\Sigma'[C] \vDash G$. Consequently, $\Sigma : \mathcal{H}_\mathcal{P},\theta \vDash G$ □

Theorem 6.3.9 (Logical Completeness). *If $\Sigma : \mathcal{H}_\mathcal{P}, C \vDash G$ then there is a finite set of constraints C_i and contexts Σ_i such that $\Sigma : G \longrightarrow^* \Sigma;\Sigma_i : C_i$ and $C \supset \bigvee_i \exists\Sigma_i[C_i]$.*

Proof. This follows from the previous theorem and the compactness of nominal logic (Corollary 5.3.3). □

6.4 Notes

The least-Herbrand-model and least-fixed-point semantics of logic programs are due originally to van Emden and Kowalski [128] and Apt and van Emden [8]. Lloyd [69], Hogger [55], and Nerode and Shore [94] are good contemporary treatments of classical logic programming semantics.

Constraint logic programming was introduced by Jaffar and Lassez [60]; Cohen [22] provides a short survey and Jaffar and Maher [61] a comprehensive and detailed survey of CLP including a discussion of semantics, existing implementations and implementation techniques, and applications. Jaffar et al. [62] is a self-contained exposition of the semantics of constraint logic programs. It may be possible to implement αProlog's functionality within an existing constraint logic programming language or framework such as CIAO Prolog [52].

Another popular form of logic programming semantics is *proof-theoretic semantics*, introduced by Miller et al. [82]. Darlington and Luo [27] and Leach, Nieva, and Rodríguez-Artalejo [68] have developed proof-theoretic semantics for Horn clause and hereditary Harrop clause constraint logic programming. The latter techniques seem particularly suitable for investigating the proof-theoretic semantics of nominal logic programming.

Chapter 7

Nominal Constraint Solving

Computers are useless. They can only give you answers.

—Pablo Picasso

The semantics in the previous chapter took a high-level view of nominal logic programming as reduction of goals to constraints, without concern for the details of constraint-solving and unification algorithms that might be used to solve the constraints. However, algorithms for solving these problems are of course necessary in a concrete implementation. In this chapter we consider these *nominal constraint solving problems*.

We first (Section 7.1) establish lower bounds on the complexity of the problems: in particular, we show that solving a single freshness, unification, or equivariant unification problem is **NP**-hard. In addition, we show that the first two problems are in **NP**; equivariant unification is conjectured to be in **NP**.

Next (Section 7.2) we supply algorithms for solving general nominal constraint problems and prove their soundness and completeness. These algorithms are high-level, and much could be done to improve their efficiency in practice (although the **NP**-hardness of nominal constraint problems indicates that it is unlikely that polynomial-time algorithms for these problems exist).

We conclude by showing that the semantics of the previous section can be combined with the constraint solving algorithms of this chapter to form a sound, complete implementation of nominal logic programming. However, such an implementation is still far from practical. We discuss the problems with the constraint solving algorithms and how they could be addressed. In particular we discuss Urban, Pitts, and Gabbay's nominal unification algorithm [126, 127], which is used in αProlog. Urban et al.'s algorithm efficiently solves a special case of nominal unification and freshness constraint problems; we refer to the terms used in this algorithm as *nominal patterns* and to the algorithm itself as *nominal pattern unification*. We also discuss some work currently in progress on identifying αProlog programs for which nominal pattern unification is complete.

7.1 Complexity

In this section, we study the complexity of solving general nominal constraints. We rely on standard definitions of the complexity classes, the concept of polynomial time reduction and **NP**-completeness, and standard **NP**-complete problems, as can be found in any complexity theory book such as Garey and Johnson [45] or Papadimitriou and Stieglitz [99].

In this section, we write $\mathbb{A}$ for a fixed name-set, $\mathbb{A} \# \mathbb{A}$ for the "fresh product" set $\{(a, b) \in \mathbb{A} \times \mathbb{A} \mid a \# b\}$, and $\mathbb{A}^{(n)}$ for the n-th iterated fresh product. Thus, $\mathbb{A}^{(n)}$ is the set $\{(a_1, \ldots, a_n) \in \mathbb{A}^{(n)} \mid a_1 \# a_2 \# \cdots \# a_n\}$. (We write $x \# y \# z \ldots$ to indicate that $x \# y, x \# z, y \# z$, etc.) Note that $\mathbb{A} \# \mathbb{A}$ and $\mathbb{A}^{(n)}$ are nominal sets.

7.1.1 Basic Problems

We write $\leq_P$ for polynomial-time reduction. Let S be a constraint problem, consisting of a subset of a specified domain $Dom(S)$ (often left implicit). We write S^* for the problem of determining simultaneous satisfiability of sets of problems from S and write $S \uplus S'$ for the problem of determining satisfiability of a single element of S or of S'. Note that $S \leq_P S^*$ and $S \leq_P S \cup T$ for any S, T, and $T \leq_P S$ for any $T \subseteq S$.

Nominal constraints are solved over the Herbrand universe of nominal terms. We use θ for a ground term valuation, and $\#_{\mathbb{NT}}$ and $=_{\mathbb{NT}}$ are (semantic) freshness and equality over nominal terms, usually abbreviated $\#$ and $=$. The *similarity* relation $t \sim u$ on nominal terms holds when t and u are equal modulo a permutation, that is,

$$t \sim u \iff \exists \pi \in \mathbb{G}. \pi \cdot t = u \ .$$

We define the following decision problems:

$$\begin{array}{rcl} \textsc{FreshSat} & = & \{a \#?\, t \mid \exists \theta. \theta(a) \# \theta(t)\} \\ \textsc{NomSat} & = & \{t \approx?\, u \mid \exists \theta. \theta(t) = \theta(u)\} \\ \text{EV}\textsc{Match} & = & \{t \precsim?\, u \mid u \text{ ground}, \exists \theta. \theta(t) \sim u\} \\ \text{EV}\textsc{Sat} & = & \{t \sim?\, u \mid \exists \theta. \theta(t) \sim \theta(u)\} \end{array}$$

Constraints of the form $t \precsim?\, u$ and $t \sim?\, u$ are called *equivariant matching* and *equivariant unification* respectively. We write S_N for the problem S restricted to name-terms.

In addition, we define the *separation* problem

$$\textsc{Sep} = \{\vec{a} \in?\, \mathbb{A}^{(n)} \mid \exists \theta. \theta(\vec{a}) \in \mathbb{A}^{(n)}\}$$

Intuitively, Sep is the set of all n-tuples of name-sorted terms that can be instantiated so that no name-constants are repeated.

Here, as in Chapter 6 and in logic programming generally, capital letters X, Y are viewed as *unknowns*.

Example 7.1.1. Here are satisfiable instances of the above five problems:

$$X \mathrel{\#?} (X\ Y) \cdot X \qquad X \approx? (X\ Y) \cdot Y \qquad \langle X\rangle (X\ Y) \cdot X \precsim? \langle \mathsf{a}\rangle \mathsf{b}$$
$$\langle X\rangle (X\ Y) \cdot X \sim? \langle Y\rangle \mathsf{b} \qquad (X, (X\ Y) \cdot X) \in? \mathbb{A}^{(2)}$$

The following instances are unsatisfiable:

$$X \mathrel{\#?} (X\ Y) \cdot Y \qquad X \approx? f((X\ Y) \cdot Y) \qquad \langle X\rangle X \precsim? \langle \mathsf{a}\rangle \mathsf{b}$$
$$\langle \mathsf{a}, \mathsf{a}, X\rangle \sim? \langle \mathsf{a}, X, \mathsf{b}\rangle \qquad (X, (\mathsf{a}\ \mathsf{b}) \cdot X, Y, (\mathsf{a}\ \mathsf{b}) \cdot Y) \in? \mathbb{A}^{(2)}$$

It is easy to see that Sep is in **NP** because a satisfying valuation θ is a polynomial-time checkable certificate for a problem in Sep. For the same reason, it is easy to see that freshness, equational, and equivariance constraint problems *involving only terms of name-sort* are in **NP**.

In addition, it is straightforward to show that nominal unification and freshness constraint solving are in **NP**.

Theorem 7.1.2. FreshSat*,* NomSat*, and* EVMatch *are in* **NP***.*

Proof. According to [127], nominal pattern unification is solvable in quadratic time. Nominal patterns constraints are constraints such that for every subterm of the form $(a\ b) \cdot t$, $\langle a\rangle t$, or $a \mathrel{\#} t$, we have a, b are name-constants. If a FreshSat or NomSat (or more generally (FreshSat $\uplus$ NomSat)*) problem P is satisfiable by θ, then let ρ be the subvaluation consisting of θ restricted to variables of name-sort. Then $\rho(P)$ is a nominal pattern since it contains no name-sorted variables at all. Hence the satisfiability of $\rho(P)$ can be verified in quadratic time. Moreover, ρ can be constructed and stored in linear time and space.

For EVMatch, note that if θ is a satisfying valuation for $s \precsim? t$, then $|\theta(s)| = |t|$, where $|\cdot|$ is the usual size measure on terms. Hence θ is itself a polynomial time verifiable certificate for $s \precsim? t$. □

However, establishing that equivariant unification is in **NP** is not so easy, because the satisfying valuation θ may be exponential in size and it is not easy to see why the witnessing permutation π should be of size polynomial in the input. We conjecture that EVSat $\in$ **NP** seems to require developing a specific nondeterministic algorithm with good space behavior. We do not believe this to be difficult in principle; however, this task is left for future work.

In the rest of this section we will show that Sep is **NP**-complete and then reduce Sep to the remaining problems. This shows that they are all **NP**-hard.

7.1.2 NP-Completeness of Separation

In the this section we prove

Theorem 7.1.3. *The problem* Sep *is* **NP***-complete.*

Proof. Having already seen SEP $\in$ **NP**, we show **NP**-hardness only. We reduce from the **NP**-complete problem GRAPH 3-COLORABILITY, that is, determining whether a graph's vertices can be colored with one of three colors so that no neighboring vertices are the same color.

Let a (directed) graph $G = (V, E)$ with n vertices and m edges be given. We assume without loss of generality that $V = \{1, \ldots, n\}$ and $E = \{e_1, \ldots, e_m\}$. We write s_i, t_i for the source and target of the edge $e_i \in E$. Let $C = \{r, g, b\}$ be a three-element subset of $\mathbb{A}$. We define a 3-coloring as an n-tuple $\vec{c} \in C^n$ such that $c_{s_i} \neq c_{t_i}$ whenever $e_i \in E$.

Define $\pi_C = (r\ g)(g\ b)$, a cyclic permutation on $\mathbb{A}$ with support C. Choose $n+m$ permutations $\tau_1, \ldots, \tau_n, \sigma_1, \ldots, \sigma_m$ so that if $T_i = \tau_i \cdot C$ for each $i \in \{1, \ldots, n\}$, and $S_j = \sigma_j \cdot C$ for each $j \in \{1, \ldots, m\}$, then all of the sets C, $T_1, \ldots, T_n$, and $S_1, \ldots, S_m$ are disjoint.

Let $X_1, \ldots, X_n \in V$ be n distinct variables.

Idea of the proof. We wish to identify colorings of G with valuations of θ satisfying some SEP instance $\vec{u}$. For such a valuation to correspond to a proper coloring, we must have $\theta(X_i) \in C$ and $\theta(X_i) \neq \theta(X_j)$ for each edge $(i, j) \in E$.

Therefore, the instance $\vec{u}$ must force all of the X_i to be elements of C and for each edge e_i force X_{s_i} and X_{t_i} to be different. In addition, $\vec{u}$ should be satisfiable whenever θ corresponds to a valid 3-coloring.

Observe $X \neq \pi_C \cdot X$ if and only if $X \in \mathit{supp}(\pi_C) = C$. So it is easy to encode a single set constraint $X \in C$ as a SEP problem

$$(X, \pi_C \cdot X) \in ?\ \mathbb{A}^{(2)}\ .$$

However, for two variables this does not quite work:

$$(X_1, \pi_C \cdot X_1, X_2, \pi_C \cdot X_2) \in ?\ \mathbb{A}^{(4)}$$

forces $X_1, X_2 \in C$ but also forces $X_1 \neq X_2$, $\pi_C \cdot X_1 \neq X_2$, etc. This is too strong. To prevent interference between subproblems, we isolate them using the permutations τ_1, τ_2:

$$(\tau_1 \cdot X_1, \tau_1 \circ \pi_C \cdot X_1, \tau_2 \cdot X_2, \tau_2 \circ \pi_C \cdot X_2) \in ?\ \mathbb{A}^{(4)}$$

First note that $\tau_1 \cdot X_1 \neq \tau_1 \circ \pi_C \cdot X_1$ implies $X_1 \neq \pi_C \cdot X_1$ so $X_1 \in C$ and similarly $X_2 \in C$, as before. On the other hand, if X_1, X_2 are in C, then all four components are different, since the first two lie in T_1 and the last two in T_2, and the two sets are disjoint. It is not hard to show by induction that

$$\vec{s} = (\tau_1 \cdot X_1, \tau_1 \circ \pi_C \cdot X_1, \ldots, \tau_n \cdot X_n, \tau_n \circ \pi_C \cdot X_n) \in ?\ \mathbb{A}^{(2n)}$$

is in SEP if and only if $X_1, \ldots, X_n \in C$.

Now we need to enforce that whenever $e_i \in E$, we have $X_{s_i} \neq X_{t_i}$. For a single edge, the following instance suffices:

$$(X_{s_i}, X_{t_i}) \in ?\ \mathbb{A}^{(2)}$$

However, as was the case earlier, problems cannot always be combined correctly because they might interfere. For example, for two edges $(1,2),(1,3)$, the problem

$$(X_1, X_2, X_1, X_3) \in ?\ \mathbb{A}^{(4)}$$

is unsatisfiable because the value of X_1 is repeated in any valuation, but $[X_1 := r, X_2 := g, X_3 := b]$ is a proper 3-coloring. To get around this problem, we use the permutations σ_i to isolate the constraints for each edge e_i. For example,

$$(\sigma_1 \cdot X_1, \sigma_1 \cdot X_2, \sigma_2 \cdot X_1, \sigma_2 \cdot X_3) \in ?\ \mathbb{A}^{(4)}$$

ensures $X_1 \neq X_2$ and $X_1 \neq X_3$. Also, if $X_1, X_2, X_3 \in C$ then the first two components are in S_1 and the second two in S_2, and $S_1 \cap S_2 = \varnothing$. So more generally, the problem

$$\vec{t} = (\sigma_1 \cdot X_{s_1}, \sigma_1 \cdot X_{t_1}, \ldots, \sigma_m \cdot X_{s_m}, \sigma_m \cdot X_{t_m}) \in ?\ \mathbb{A}^{(2m)}$$

enforces the coloring property for each edge and permits all valid colorings.

Define $\vec{u}$ to be the $2n + 2m$-tuple $\vec{s}\vec{t}$. Then $\vec{u} \in ?\ \mathbb{A}^{(2n+2m)}$ is the Sep problem corresponding to the instance G of Graph 3-Colorability.

Correctness of the reduction. So far we have only described the construction and the intuition behind it. It is easy to see that $\vec{u}$ can be constructed in $O(m+n)$ time, since π_C, τ_i, and σ_j each have representations consisting of at most three transpositions. We now show carefully that the reduction is correct, that is, G has a 3-coloring $\vec{c} \in C^n$ if and only if $\vec{u}$ has a separating valuation θ. The backward direction is easy, since (as outlined above) it is easy to show that any solution θ separating $\vec{u} = \vec{s}\vec{t}$ corresponds to a 3-coloring $c_i = \theta(X_i)$.

The difficulty is showing that $\vec{u}$ is not over-constrained: that is, if $\vec{c}$ is a 3-coloring then the valuation $\theta(X_i) = c_i$ separates $\vec{u}$. Suppose $\vec{c}$ is a 3-coloring and $\theta(X_i) = c_i$. We need to show that $i \neq j$ implies $\theta(u_i) \neq \theta(u_j)$ for each $i, j \in \{1, \ldots, |\vec{u}|\}$. Assume $i, j \in \{1, \ldots, |\vec{u}|\}$ and $i \neq j$. Suppose without loss of generality that $i < j$. There are three cases.

If i is even or $j > i+1$, then $u_i = \rho \cdot X_k$ and $u_j = \rho' \cdot X_{k'}$ for some permutations ρ, ρ' and $X_k, X_{k'}$, and $\rho \cdot C$ and $\rho' \cdot C$ are disjoint, so

$$\theta(u_i) = \rho \cdot c_k \neq \rho' \cdot c_{k'} = \theta(u_j)$$

If i is odd and $i + 1 = j$ and $j \leq 2n$, then j is even; set $k = j/2$. Then $u_i = \tau_k \cdot X_k, u_j = \tau_k \circ \pi_C \cdot X_k$, and we have

$$\theta(u_i) = \tau_k \cdot c_k \neq \tau_k \circ \pi_C \cdot c_k = \theta(u_j)$$

since $\pi_C \cdot c_k \neq c_k$.

If i is odd and $j = i + 1$ and $2n + 1 \leq i$, then j and $j - 2n$ are even; set $k = (j - 2n)/2$. Then $u_i = \sigma_k \cdot X_{s_k}$, $u_j = \sigma_k \cdot X_{t_k}$, and

$$\theta(u_i) = \sigma_k \cdot c_{s_k} \neq \sigma_k \cdot c_{t_k} = \theta(u_j)$$

where $c_{s_k} \neq c_{t_k}$ since c is a 3-coloring. So, in any case, $\theta(u_i) \neq \theta(u_j)$. QED. $\square$

7.1.3 Freshness Constraint Satisfaction

In this section we reduce the separation problem to freshness constraint. The reduction takes two steps: first we show that a single instance of SEP reduces to a set of freshness constraints, then reduce such sets of constraints to single general constraints.

Theorem 7.1.4. SEP $\leq_P$ FRESHSAT$^*_N$ $\leq_P$ FRESHSAT

Proof. For SEP $\leq_P$ FRESHSAT$^*_N$, let $\vec{a}$ be a SEP problem instance, that is, a sequence of terms ν for which it is desired to find a valuation θ satisfying $\theta(a_i) \# \theta(a_j)$ whenever $i \neq j$. Then consider the set

$$S = \{a_i \#? \, a_j \mid i \neq j\}$$

This set is an instance of FRESHSAT$^*_N$ and clearly a solution θ for $\vec{t} \in$ SEP is a solution for $S \in$ FRESHSAT*. The size of S is quadratic in the size of $\vec{a}$, so S can be constructed from $\vec{a}$ in polynomial time.

For the second reduction FRESHSAT$^* \leq$ FRESHSAT, let $S = \{a_1 \# t_1, \dots, a_n \# t_n\}$ be an instance of FRESHSAT*. Let b be a name not appearing in S. Consider the problem

$$\mathsf{b} \#? \, ((\mathsf{b}\ a_1) \cdot t_1, \dots, (\mathsf{b}\ a_n) \cdot t_n) \tag{7.1}$$

Clearly the above problem can be constructed from S in linear time. We show that θ satisfies (7.1) if and only if θ satisfies S.

Suppose θ satisfies (7.1), that is, $\mathsf{b} \# \theta((\mathsf{b}\ a_1) \cdot t_1, \dots, (\mathsf{b}\ a_n) \cdot t_n)$. Then $\mathsf{b} \# (\mathsf{b}\ \theta(a_i)) \cdot \theta(t_i)$ for each i, and so $\theta(a_i) \# \theta(t_i)$, so $\theta(S)$ holds. Conversely, suppose $\theta(S)$ holds; then $\theta(a_i) \# \theta(t_i)$ for each i and so $\mathsf{b} \# (\mathsf{b}\ \theta(a_i)) \cdot \theta(t_i)$ for each i. Consequently $\mathsf{b} \# \theta(\vec{t})$. □

Corollary 7.1.5. FRESHSAT *is* **NP**-*hard.*

7.1.4 Nominal Equational Satisfaction

In this section we consider the reduction of general freshness problems to general nominal equational constraints. Recall that the following is a theorem of NL:

$$a \# t \iff \text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot t \approx t$$

This reduces freshness constraint solving to solving quantified problems of the form $\text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot t \approx t$. This is somewhat circular, since $\text{И}\mathsf{b}.(a\ \mathsf{b}) \cdot t \approx t \iff \mathsf{b} \# a, t \wedge (a\ \mathsf{b}) \cdot t \approx t$ for some b not syntactically present in a, t. However, even if b is syntactically fresh for a and t, the single constraint $(a\ \mathsf{b}) \cdot t \approx t$ is not equivalent to $a \# t$, since there is the possibility that $a \approx \mathsf{b}$ holds. The way we deal with this is to use two equality constraints with fresh names b and b', specifically,

$$(a\ \mathsf{b}) \cdot t \approx t \wedge (a\ \mathsf{b}') \cdot t \approx t$$

Since a cannot be equal to both b and b', this constraint ensures that $a \# t$.

Proposition 7.1.6. *Let $a, t : \nu$ be given and $\mathsf{b}, \mathsf{b}' \notin FN(a,t)$. Then $a \# t$ is satisfiable if and only if $(a\ \mathsf{b}) \cdot t \approx t \wedge (a\ \mathsf{b}') \cdot t \approx t$ is satisfiable.*

Proof. For the forward direction, suppose $\theta \vDash a \# t$. Without loss of generality, we may assume that θ does not mention b or b', since otherwise we may rename θ to obtain $\theta' \vDash a \# t$ because neither a nor t mentions b or b'. Then $\theta \vDash \mathsf{b} \# t$ and $\theta \vDash \mathsf{b}' \# t$ so $\theta \vDash (a\ \mathsf{b}) \cdot t \approx t \wedge (a\ \mathsf{b}') \cdot t \approx t$.

For the reverse direction, suppose $\theta \vDash (a\ \mathsf{b}) \cdot t \approx t \wedge (a\ \mathsf{b}') \cdot t \approx t$. Then $\theta \vDash (a\ \mathsf{b}) \cdot t \approx t$ and $\theta \vDash (a\ \mathsf{b}') \cdot t \approx t$. Since $a, t : \nu$, there are only three cases:

1. $\theta \vDash a \approx \mathsf{b}$ and $\theta \vDash a \approx \mathsf{b}'$. This case is impossible since $\mathsf{b} \neq \mathsf{b}'$, so vacuously $\theta \vDash a \# t$.
2. $\theta \vDash a \# \mathsf{b}$. Then since $\theta \vDash (a\ \mathsf{b}) \cdot t \approx t$, we must have $\theta \vDash a \# t$.
3. $\theta \vDash a \# \mathsf{b}'$. Symmetric to (2).

So, in any case, $\theta \vDash a \# t$. □

This is the key to the reduction from FRESHSAT$^*_N$ to NOMSAT$^*_N$.

Theorem 7.1.7. FRESHSAT$^*_N \leq_P$ NOMSAT$^*_N \leq_P$ NOMSAT.

Proof. By Proposition 7.1.6, each element $a \# t \in S$ of $S \in$ FRESHSAT$^*_N$ can be transformed to constraints $(a\ \mathsf{b}) \cdot t \approx ?\ t, (a\ \mathsf{c}) \cdot t \approx ?\ t$, where $\mathsf{b}, \mathsf{b}' \notin S$, yielding an instance S' of NOMSAT. This transformation preserves satisfiability by the previous lemma.

For the second reduction, we transform the problem $S = \{t_1 \approx ?\ u_1, \ldots, t_n \approx ?\ u_n\}$ to $(t_1, \ldots, t_n) \approx ?\ (u_1, \ldots, u_n)$, which is obviously equivalent. □

Corollary 7.1.8. NOMSAT *is* **NP**-*hard.*

7.1.5 Equivariant Matching and Satisfaction

Finally, we show that SEP reduces to EVSAT. In fact, SEP can be reduced to the seemingly simpler problem of equivariant matching, that is, equivariant unification problems with one side ground. To prove this, we need the following obvious fact about $\mathbb{A}^{(n)}$.

Lemma 7.1.9. *If $\vec{\mathsf{a}}, \vec{\mathsf{b}} \in \mathbb{A}^{(n)}$, then there exists a permutation π such that $\pi \cdot \vec{\mathsf{a}} = \vec{\mathsf{b}}$.*

Theorem 7.1.10. SEP $\leq_P$ EVMATCH $\leq_P$ EVSAT

Proof. Choose some $\vec{\mathsf{b}} \in \mathbb{A}^{(n)}$. For SEP $\leq_P$ EVMATCH, let $\vec{a} \in ?\ \mathbb{A}^{(n)}$ be an instance of SEP. We transform the SEP-instance to $\vec{a} \sim ?\ \vec{\mathsf{b}}$. This reduction requires linear time. We must show that $\vec{a} \in ?\ \mathbb{A}^{(n)} \in$ SEP if and only if $\vec{a} \sim ?\ \vec{\mathsf{b}} \in$ EVMATCH. For the forward direction, assume $\theta(\vec{a}) \in \mathbb{A}^{(n)}$. By the lemma, we have $\pi \cdot \theta(\vec{a}) = \vec{\mathsf{b}}$, so $\theta(\vec{a}) \sim \vec{\mathsf{b}}$. Conversely, if $\pi \cdot \theta(\vec{a}) = \vec{\mathsf{b}}$, where $\vec{\mathsf{b}} \in \mathbb{A}^{(n)}$, it follows that $\theta(\vec{a}) = \pi^{-1} \cdot \pi \cdot \theta(\vec{a}) \in \mathbb{A}^{(n)}$ since the latter is closed under swapping.

For EVMATCH $\leq_P$ EVSAT, the reduction is by inclusion. □

Corollary 7.1.11. EVMATCH *and* EVSAT *are* **NP**-*hard.*

7.2 Algorithms

7.2.1 Abstract Constraint Solving Algorithms

We now define constraint-solving algorithms at an abstract level.

Definition 7.2.1 (Abstract constraint solving algorithms). *A* constraint solving problem *$(\mathcal{P}, \mathcal{S})$ is a set of problems $\mathcal{P}$ and a subset $\mathcal{S} \subseteq \mathcal{P}$ called the set of* solved forms. *A* problem instance *$\Sigma : P$ consists of a context Σ and set P of constraints, well-formed with respect to Σ. A* constraint solving algorithm *is a triple $\mathcal{A} = \langle \mathcal{P}, \mathcal{S}, \longrightarrow \rangle$, where $\longrightarrow : \mathcal{P} \times \mathcal{P}$ is a rewriting relation on problems in $\mathcal{P}$.*

We say that a constraint problem S is solved *if $S \in \mathcal{S}$, and S is a* solution *for P if $P \longrightarrow^* S$ and S is solved. We write $Solv(P)$ for the set of solutions for P.*

We say that a constraint set P is stuck *if no transition can be taken from P and $P \notin \mathcal{S}$.*

For example, a constraint solving algorithm over numerical constraints might include rules like $\{x + 1 \approx 5\} \uplus P \longrightarrow \{x \approx 4\} \uplus P$, and the solved forms might be constraint sets in which every equation is of the form $\{x \approx n\}$ and no variable occurs twice. Note that $Solv(P)$ may be infinite. Also, the choice of solved forms is arbitrary: we do not require that solved forms be satisfiable, although this property is desirable. This frees us to consider so-called pre-unification or partial constraint solving algorithms that reduce complex constraints to simpler, but not necessarily satisfiable constraints.

Lemma 7.2.2. *If $P \longrightarrow P'$ then $Solv(P') \subseteq Solv(P)$. Moreover, $Solv(P) = \bigcup\{Solv(Q) \mid P \longrightarrow Q\} \cup (P \cap \mathcal{S})$.*

Definition 7.2.3 (Properties of abstract constraint solving algorithms). *We say that an algorithm is* sound *for P if $P \longrightarrow^* P'$ implies $P' \vDash P$, and* complete *for P if $P \vDash \bigvee Solv(P)$. An algorithm is* terminating *if there are no infinite reduction sequences. An algorithm is* progressive *if whenever P is not in solved form and $\theta \vDash P$ there exists a Q such that $P \longrightarrow Q$ and $\theta \vDash Q$.*

The soundness and completeness properties are directly analogous to the respective properties for a logic or operational semantics. Note that soundness and completeness together imply $P \iff \bigvee Solv(P)$.

Termination and progress are important considerations for a constraint solving algorithm. Algorithms that satisfy all four properties are particularly well behaved. In fact, a terminating, progressive algorithm is complete:

Theorem 7.2.4. *If $\mathcal{A}$ is terminating and progressive, then it is also complete.*

Proof. Consider the graph $T = (\mathcal{P}, \longrightarrow)$, that is, whose nodes are problems and whose edges are transitions. Since $\mathcal{A}$ is terminating, this graph is a well-founded tree.

We prove that $P \vDash \bigvee Solv(P)$ by well-founded induction on T. If P is stuck then (by the progress property) it is unsatisfiable so $P \vDash \bigvee Solv(P)$ vacuously. If P is in solved form, $P \in Solv(P)$ so $P \vDash \bigvee Solv(P)$.

Otherwise, consider $\mathcal{Q} = \{Q \mid P \longrightarrow Q\}$, the set of all children of P. By induction, $Q \vDash \bigvee Solv(Q)$ for each $Q \in \mathcal{Q}$. Moreover, $Solv(P) = \bigcup_{Q \in \mathcal{Q}} Solv(Q) \cap (P \cup \mathcal{S})$, so $\bigvee \mathcal{Q} \vDash Solv(P)$. To show $P \vDash Solv(P)$, it suffices to show that $P \vDash \bigvee \mathcal{Q}$.

Suppose $\theta \vDash P$. Then P can take a step to some $Q \in \mathcal{Q}$, $P \longrightarrow Q$, such that $\theta \vDash Q$. Hence, $\theta \vDash \bigvee \mathcal{Q}$. Since θ was arbitrary, $P \vDash \bigvee Q$. □

7.2.2 Name-Name Constraints

We first focus on the restricted language of name-name constraints:

$$\begin{array}{rcl} a & ::= & \mathsf{a} \mid X \mid (a\ b) \cdot a' \\ C & ::= & a \mathrel{\#} b \mid a \approx b \end{array}$$

As shown in Section 7.1, constraint solving for sets of name-name constraints is an **NP**-complete problem, so it is unlikely that an efficient algorithm exists for reducing arbitrary name-name constraints to a satisfiable solved form if one exists. We present, instead, a complete algorithm that reduces such constraints to satisfiable solved forms.

Definition 7.2.5. *We say a name-name problem is* solved *if:*

1. *All equations are of the form $X \approx t$, where X appears nowhere else in the problem.*

2. *All freshness constraints are of the form $a \mathrel{\#} b$, where a, b are distinct variables or name-constants.*

Thus, $(\mathsf{a}\ \mathsf{b}) \cdot X \approx Y$ is not solved, but $X \approx \mathsf{a}, Y \approx \mathsf{b}$ is; $A \mathrel{\#} (A\ B) \cdot A$ is not solved, but $A \mathrel{\#} B$ is.

Proposition 7.2.6. *Every solved name-name problem is satisfiable.*

Proof. We prove this in two stages. First we show that every solved problem including only freshness constraints is satisfiable. Then we generalize this to show how to find a valuation for any solved problem.

For the first part, let S be the solved problem. Let $\vec{X} = FV(S)$ and $\vec{\mathsf{a}} = FN(S)$. Let $\vec{\mathsf{b}}$ be freshly chosen names, one for each free variable in $\vec{X}$, and let θ be the valuation mapping X_i to b_i for each i. Note that this ensures that if $a \neq b$ then $\theta(a) \neq \theta(b)$. We will show that $\theta \vDash S$ is satisfiable by induction on the size of S. If S is empty, obviously $\theta \vDash S$. If $S = S' \cup a \mathrel{\#?} b$ then by induction we have $\theta \vDash S'$. Moreover, since $a \neq b$ we have $\theta(a) \neq \theta(b)$, i.e. $\theta(a) \mathrel{\#} \theta(b)$. Hence $\theta \vDash S' \cup \{a \mathrel{\#?} b\}$.

For the second part, the proof is by induction on the number of equational constraints in the problem. In the base case, there are no equality formulas so

Table 7.1: Name-name constraint solving rules

$(\#_1)$	$\{\mathsf{a} \mathrel{\#?} \mathsf{b}\} \uplus P$	$\longrightarrow$	P
$(\#_2)$	$\{(a\ a') \cdot b \mathrel{\#?} b'\} \uplus P$	$\longrightarrow$	$\{a \approx? b, a' \mathrel{\#?} b'\} \uplus P$
$(\#_3)$	$\{(a\ a') \cdot b \mathrel{\#?} b'\} \uplus P$	$\longrightarrow$	$\{a \mathrel{\#?} b, a' \mathrel{\#?} b, b \mathrel{\#?} b'\} \uplus P$
$(\approx_1)$	$\{t \approx? t\} \uplus P$	$\longrightarrow$	P
$(\approx_2)$	$\{(a\ a') \cdot b \approx? b'\} \uplus P$	$\longrightarrow$	$\{b \approx? a, b' \approx? a'\} \uplus P$
$(\approx_3)$	$\{(a\ a') \cdot b \approx? b'\} \uplus P$	$\longrightarrow$	$\{a \mathrel{\#?} b, a' \mathrel{\#?} b, b \approx? b'\} \uplus P$
$(\approx_4)$	$\{a \approx? \pi \cdot X\} \uplus P$	$\longrightarrow$	$P[\pi^{-1} \cdot a/X] \uplus \{X \approx? \pi^{-1} \cdot a\}$ $X \notin FV(\pi, a)$

the first part applies. In the inductive case, let $P = \{X \approx a\} \uplus P'$, where $X \notin FV(P', a)$. By induction, there is a valuation $\theta \vDash P'$ Without loss of generality, assume $X \notin dom(\theta)$. Because $X \notin FV(P', t)$, this valuation can be extended to θ' that assigns values to all the variables of t, and assigns X the value $\theta(a)$. □

Now we present a reduction system for solving name-name constraint problems. We consider constraints equivalent up to symmetry with respect to $\#?$ and $\approx?$, and consider terms equivalent up to reordering the arguments of transpositions (that is, $(a\ a') \cdot t = (a'\ a) \cdot t$). These conventions greatly simplify the transition system and proofs, and are not difficult to deal with in an implementation by adding the symmetric cases.

The reductions are shown in Table 7.1. The context Σ never changes in any of the transitions, so is omitted. The $(\#_1)$ and $(\approx_1)$ rules indicate that trivially solvable constraints can be removed. The $(\#_{2,3})$ and $(\approx_{2,3})$ rules decompose swappings by case distinction. The $(\approx_4)$ rule performs variable elimination in the case that the variable does not occur in the relevant permutation. This rule can be applied eagerly to avoid nondeterminism due to swappings whenever possible.

Example 7.2.7. These rules can be used to derive all possible solutions to the following name-name constraints. For example, $\{A \approx b, X \approx Y\}$ is already in solved form, whereas $\{A \approx Y, X \approx (A\ \mathsf{b}) \cdot Y\}$ can be reduced to several solved forms:

$$\{A \approx Y, X \approx (Y\ \mathsf{b}) \cdot Y\} \qquad \{A \approx \mathsf{b}, X \approx \mathsf{b}\} \qquad \{A \approx Y, X \approx \mathsf{b}\}$$

Note that the first solution is a generalization of the second and third, obtained by variable-elimination on A. A fourth possible solution, $\{A \approx Y, X \approx Y, \mathsf{b} \# Y, Y \# Y\}$, is ruled out because of the unsatisfiable constraint $Y \# Y$.

Theorem 7.2.8. *NNU is terminating.*

Proof. We will exhibit a decreasing, well-founded measure on problems P.

Let $\mu(t)$ be defined as follows:

$$\begin{aligned} \mu(a) = \mu(X) &= 0 \\ \mu((a\ b) \cdot t) &= \mu(a) + \mu(b) + 3\mu(x) + 1 \\ \mu(t \mathrel{\#} u) = \mu(t \approx u) &= \mu(t) + \mu(u) \end{aligned}$$

Let $n_1 = |FV(P)|$ be the number of variables remaining in P, $n_2 = \sum_{C \in P} \mu(P)$, and $n_3 = ||P||$, the total size of all the terms in P. Consider the measure $\nu(P) = (n_1, n_2, n_3)$, on $\mathbb{N}^3$ ordered lexicographically. We will show that each transition of NNU decreases ν. The following table summarizes the behavior of the transitions with respect to n_1, n_2, n_3:

	$(\#_1)$	$(\#_2)$	$(\#_3)$	$(\approx_1)$	$(\approx_2)$	$(\approx_3)$	$(\approx_4)$
n_1	$=$	$=$	$=$	$\geq$	$=$	$=$	$>$
n_2	$=$	$>$	$>$	$=$	$>$	$>$	
n_3	$>$			$>$			

The interesting cases are $(\#_3)$ and $(\approx_3)$; they are similar. Consider $(\#_3)$:

$$Q_1 = \{b \mathrel{\#?} (a\ a') \cdot b'\} \uplus P \longrightarrow \{a \mathrel{\#?} b', a' \mathrel{\#?} b', b \mathrel{\#?} b'\} \uplus P = Q_2$$

Both sides have the same number of free variables, so n_1 remains the same. To see why, note that

$$\begin{aligned} n_2(Q_1) &= \mu(b) + \mu(a) + \mu(a') + 3\mu(b') + 1 + \mu(P) \\ n_2(Q_2) &= \mu(a) + \mu(b') + \mu(a') + \mu(b') + \mu(b) + \mu(b') + \mu(P) \end{aligned}$$

Clearly, $n_1(Q_2) - n_2(Q_2) = 1$, so n_2 decreases in a $(\#_3)$ transition. □

Theorem 7.2.9. *NNU is sound.*

Proof. It suffices to show that each rewriting rule is sound: that is, if $P \longrightarrow P'$ then $P' \vDash P$. For $(\#_1)$ and $(\approx_1)$, this is obvious.

For $(\#_{2,3})$ and $(\approx_{2,3})$, each implication is easy to prove using properties of nominal logic. For example, for $(\#_2)$, we need to show $P, a \approx b, a' \mathrel{\#} b' \vDash P, b \mathrel{\#} (a\ a') \cdot b'$. Since $a \approx b$, we have $a' \approx (a\ a') \cdot a \approx (a\ a') \cdot b$, so $(a\ a') \cdot b \mathrel{\#} b'$, and by equivariance $b \mathrel{\#} (a\ a') \cdot b$. The other proofs are similar.

Finally, for $(\approx_4)$, we need to show $P[\pi^{-1} \cdot t/X], X \approx \pi^{-1} \cdot t \vDash P, \pi \cdot X \approx t$. Using the substitution principle to reverse the substitution of $\pi^{-1} \cdot t$ for X, we can derive P from $P[\pi^{-1} \cdot t/X]$, and we can prove $\pi \cdot X \approx t$ from $X \approx \pi^{-1} \cdot t$ using equivariance several times to invert π. □

Proposition 7.2.10. *NNU is progressive.*

Proof. The proof is by induction on the size of P and construction of name-name constraints $a \mathrel{\#?} b$, $a \approx? b$. We prove that for $\theta \vDash P$, if P is not solved then for some Q, $P \longrightarrow Q$ and $\theta \vDash Q$.

Assume $\theta \vDash P$. If P is empty then P is already solved. Otherwise, there are several cases.

- If $P = \{a \mathrel{\#?} b\} \uplus P'$ then there are several further cases.
  - If $a = (a_1\ a_2) \cdot a_3$, then there are (up to symmetry) two sub-cases:
    - If $\theta(a_1) = \theta(a_3)$ and $\theta(a_2) \mathrel{\#} \theta(b)$, then $\theta \vDash Q = \{a_1 \approx? a_3, a_2 \mathrel{\#?} b\} \uplus P'$, and $P \longrightarrow_{\#_2} Q$.
    - If $\theta(a_1) \mathrel{\#} \theta(a_3)$, $\theta(a_2) \mathrel{\#} \theta(a_3)$, and $\theta(a_3) \mathrel{\#} \theta(b)$, then $\theta \vDash Q = \{a_1 \mathrel{\#} a_3, a_2 \mathrel{\#?} a_3, a_3 \mathrel{\#?} u\} \uplus P'$. Moreover, $P \longrightarrow_{\#_3} Q$.
  - The case for $b = (b_1\ b_2) \cdot b_3$ is symmetric.
  - Otherwise a and b are both names or variables. Since $a \mathrel{\#?} b$ is satisfiable, a and b are different terms. If a and b are both names, then $(\#_1)$ applies, and obviously $\theta \vDash P'$. Otherwise, $a \mathrel{\#?} b$ is in solved form, and we may proceed by induction on P'.
- If $P = \{a \approx? b\} \uplus P'$, then there are several further cases.
  - If a is of the form $(a_1\ a_2) \cdot a_3$, then there are (up to symmetry) two sub-cases:
    - If $\theta(a_1) = \theta(a_3)$ and $\theta(a_2) = \theta(b)$, then $\theta \vDash Q = \{a_1 \approx? a_3, a_2 \approx? b\} \uplus P'$, and $P \longrightarrow_{\approx_2} Q$.
    - If $\theta(a_1) \mathrel{\#} \theta(a_3)$, $\theta(a_2) \mathrel{\#} \theta(a_3)$, and $\theta(a_3) \mathrel{\#} \theta(b)$, then $\theta \vDash Q = \{a_1 \mathrel{\#?} a_3, a_2 \mathrel{\#?} a_3, a_3 \approx? b\} \uplus P'$. Moreover, $P \longrightarrow_{\approx_3} Q$.
  - The case for $u \approx? (a_1\ a_2) \cdot a_3$ is symmetric.
  - Otherwise, a and b are names or variables. If $a = X = b$, then $(\approx_1)$ applies. If $a \neq b$, then one of a, b must be a variable, because equations among distinct names are unsatisfiable. Without loss, assume $a = X$. Clearly, $X \notin FV(b)$ since $a \neq b$. If X appears in P' then $\theta \vDash Q = \{X \approx? b\} \uplus P'\{\mathsf{a}/X\}$, and $P \longrightarrow_{\approx_4} Q$. Otherwise, $X \approx? \mathsf{a}$ is in solved form relative to P', so we may proceed by induction.

This analysis exhausts all cases and completes the proof. □

Corollary 7.2.11. *NNU is complete.*

7.2.3 Freshness Constraints

We now consider solving freshness constraints $a \mathrel{\#?} t$, where $a : \nu$ and $t : \sigma$ are well-formed nominal terms. To simplify matters, we assume that terms t are kept

Table 7.2: Freshness constraint solving

$\Sigma : \{a \mathrel{\#?} \langle\rangle\} \uplus P$	$\longrightarrow$	$\Sigma : P$
$\Sigma : \{a \mathrel{\#?} f(t)\} \uplus P$	$\longrightarrow$	$\Sigma : \{a \mathrel{\#?} t\} \uplus P$
$\Sigma : \{a \mathrel{\#?} \langle t_1, t_2\rangle\} \uplus P$	$\longrightarrow$	$\Sigma : \{a \mathrel{\#?} t_1, a \mathrel{\#?} t_2\} \uplus P$
$\Sigma : \{a \mathrel{\#?} \langle b\rangle t\} \uplus P$	$\longrightarrow$	$\Sigma\#\mathsf{c} : \{a \mathrel{\#?} (b\ \mathsf{c}) \cdot t\} \uplus P$

in a normal form with swappings pushed down as far as possible past function symbols, abstractions, and constants, that is, where a and t are of the following forms:

$$\begin{array}{rcl} a & ::= & \mathsf{a}^\nu \mid (a\ a') \cdot b \mid X^\sigma \\ t & ::= & a \mid f(t) \mid \langle t_1, t_2\rangle \mid \langle\rangle \mid \langle a\rangle t \end{array}$$

Note that variables X may be of name-sort or other sorts. General nominal terms can be normalized to the above forms using the following rewriting rules:

$$\begin{array}{rcl} (a\ b) \cdot \langle\rangle & \to & \langle\rangle \\ (a\ b) \cdot f(t) & \to & f((a\ b) \cdot t) \\ (a\ b) \cdot \langle t_1, t_2\rangle & \to & \langle (a\ b) \cdot t_1, (a\ b) \cdot t_2\rangle \\ (a\ b) \cdot \langle a'\rangle t & \to & \langle (a\ b) \cdot a'\rangle (a\ b) \cdot t \end{array}$$

General freshness constraint solving problems are sets of freshness constraints over arbitrary (normalized) terms. For such problems, the appropriate solved forms are constraints involving only names, variables, and swapping. Among such constraints, name-name constraints can be further solved using the algorithm of the previous section, whereas constraints involving non-name variables X can always be put into the form $a \mathrel{\#?} X$. Such constraints are always satisfiable.

Definition 7.2.12. *A freshness constraint is said to be in* solved form *if*

1. *It is of the form $a \mathrel{\#?} b$, where $a : \nu, b : \nu'$ are of name-sort, or*
2. *It is of the form $a \mathrel{\#?} X$, where $a : \nu$ and X is not of name-sort.*

We now show how to reduce general freshness constraints to solved forms. We propose a set of reduction rules for solving freshness constraints as shown in Table 7.2. We define FCS to be the constraint solving algorithm defined by these rules.

Example 7.2.13. Here is a simple example of freshness constraint solving:

$$\begin{array}{rcl} \Sigma : \{A \# \langle A\rangle\langle A, X\rangle\} & \longrightarrow & \Sigma\#\mathsf{c} : \{A \# \langle (A\ \mathsf{c}) \cdot A, (A\ \mathsf{c}) \cdot X\rangle\} \\ & \longrightarrow & \Sigma\#\mathsf{c} : \{A \# (A\ \mathsf{c}) \cdot A, A \# (A\ \mathsf{c}) \cdot X\} \\ & \longrightarrow & \Sigma\#\mathsf{c} : \{A \# (A\ \mathsf{c}) \cdot A, (A\ \mathsf{c}) \cdot A \# X\} \end{array}$$

Note that the remaining constraint set is valid, intuitively because $A \# (A\ \mathsf{c}) \cdot A$ reduces to $\mathsf{c} \# A$ and similarly $\mathsf{c} \# X$ is equivalent to the second constraint. Since c is fresh, both constraints are valid.

We now show that *FCS* is a sound, complete algorithm for solving freshness constraints.

Lemma 7.2.14. *The following formulas are valid in the Herbrand universe of nominal terms:*

$$a \# \langle\rangle \tag{7.2}$$

$$a \# t \iff a \# f(t) \tag{7.3}$$

$$a \# t_1 \wedge a \# t_2 \iff a \# \langle t_1, t_2\rangle \tag{7.4}$$

$$\text{И}\mathsf{c}.a \# (b\ \mathsf{c}) \cdot t \iff a \# \langle a\rangle t \tag{7.5}$$

Proof. The forward directions are theorems of NL. The reverse directions of the first three are standard, and the reverse direction of (7.5) is a theorem of NL. □

Theorem 7.2.15. *FCS is sound.*

Proof. We must show that for each transition $P \longrightarrow P'$, we have $P' \vDash P$. These facts follow from (7.2)–(7.5) respectively. □

Proposition 7.2.16. *FCS is terminating.*

Proof. To prove this fact, we define a measure on freshness constraint problems P which decreases at each reduction step. Let

$$\begin{array}{rcl} \mu(a) & = & 0 \\ \mu(\langle\rangle) & = & 1 \\ \mu(\langle t_1, t_2\rangle) & = & \mu(t_1) + \mu(t_2) + 1 \\ \mu(f(t)) & = & \mu(t) + 1 \\ \mu(\langle a\rangle t) & = & \mu(t) + 1 \end{array}$$

In words, $\mu(t)$ is the number of term symbols in t excluding swapping, names, and variables. Note that μ places no weight on the names involved in swappings, that is, $\mu((a\ b) \cdot t) = \mu(t)$.

It is easy to see that $\sum_{(a\#?t)\in P} \mu(a) + \mu(t)$ decreases after each reduction step in *FCS*. □

Proposition 7.2.17. *FCS is progressive.*

Proof. We prove that if $\theta \vDash P$ then there exists a Q such that $P \longrightarrow Q$ and $\theta \vDash Q$ by induction on P. Let $\theta \vDash P$ be given. If P is empty then the result is vacuous. Otherwise, there are several cases.

- If $P = \{a \#?\ \langle\rangle\} \uplus P'$ then $P \longrightarrow P'$ and clearly $\theta \vDash P'$ by (7.2).

- If $P = \{a \mathrel{\#?} f(t)\} \uplus P'$ then $P \longrightarrow \{a \mathrel{\#?} t\} \uplus P'$ and $\theta \vDash \{a \mathrel{\#?} t\} \uplus P'$ by (7.3).
- If $P = \{a \mathrel{\#?} \langle t_1, t_2 \rangle\} \uplus P'$ then $P \longrightarrow \{a \mathrel{\#?} t_1, a \mathrel{\#?} t_2\} \uplus P'$. Moreover, $\theta \vDash \{a \mathrel{\#?} t_1, a \mathrel{\#?} t_2\} \uplus P'$ by (7.4).
- Finally, if $P = \Sigma : \{a \mathrel{\#?} \langle b \rangle t\} \uplus P'$ then $P \longrightarrow \Sigma\#\mathsf{c} : \{a \mathrel{\#?} (b\ \mathsf{c}) \cdot t\} \uplus P'$. Also, by (7.5), we have $\Sigma : \theta \vDash \text{И}\mathsf{c}.\ \#?\ (b\ \mathsf{c}) \cdot t, P$, so $\Sigma\#\mathsf{c} : \theta \vDash \{a \mathrel{\#?} (b\ \mathsf{c}) \cdot t\} \uplus P'$.

This analysis exhausts all cases and completes the proof. □

Corollary 7.2.18. *FCS is complete.*

7.2.4 Nominal Unification

Next we consider the problem of solving equations among nominal terms, that is, *nominal unification.* As in the previous section, we require terms to be normalized so that swappings only occur around variables, names, or other swappings.

In ordinary unification, solved forms are sets S of equations $X \approx? t$ such that X does not appear in t or $S - (X \approx? t)$. For nominal unification, it is necessary to generalize this definition.

In ordinary unification, when we encounter a problem of the form $X \approx? t$, there are three cases. If $t = X$, then the problem is trivial; otherwise, if t contains X, there is no solution; finally, if t does not contain X then we may eliminate X by substituting for t. However, in nominal unification, some equations of the form $X = t(X)$ have solutions. For example $X = (W\ Z) \cdot X$ has several possible solutions, including $Z \approx W$ and $Z \mathrel{\#} X, W \mathrel{\#} X$. It seems difficult to search for all possible solutions to such a problem because complex terms may be substituted for X. However, if X is otherwise unconstrained, then a solution always exists. Therefore, we permit equations of the form $\pi \cdot X \approx? \pi' \cdot X$ within solutions.

Definition 7.2.19. *A* solved form *S is a set of equations, each of which satisfies*

1. *C is a name-name equation, or*
2. *$C = \pi_1 \cdot X \approx? \pi_2 \cdot X$, or*
3. *$C = X \approx? t$, where X does not appear in t or elsewhere in $S - (X \approx? t)$.*

Note that a constraint set like $\{X \approx? \pi \cdot X, X \approx? \langle\rangle\}$ is not in solved form since X can be eliminated to get the problem $\{\langle\rangle \approx? \langle\rangle, X \approx? \langle\rangle\}$, which simplifies to $\{X \approx? \langle\rangle\}$. Table 7.3 lists the transitions of an algorithm *NU* for solving nominal unification problems. The first three rules and fifth rule are standard unification steps. The fourth rule unifies two abstractions by generating a fresh name c and swapping the names a and b respectively with c in the bodies of the abstractions t and u, unifying the result.

Table 7.3: Nominal unification

$\Sigma : \{\langle\rangle \approx ? \langle\rangle\} \uplus P$	$\longrightarrow$	$\Sigma : P$
$\Sigma : \{f(t) \approx ? f(u)\} \uplus P$	$\longrightarrow$	$\Sigma : \{t \approx ? u\} \uplus P$
$\Sigma : \{\langle t_1, t_2\rangle \approx ? \langle u_1, u_2\rangle\} \uplus P$	$\longrightarrow$	$\Sigma : \{t_1 \approx ? u_1, t_2 \approx ? u_2\} \uplus P$
$\Sigma : \{\langle a\rangle t \approx ? \langle b\rangle u\} \uplus P$	$\longrightarrow$	$\Sigma\#\mathsf{c} : \{(a\ \mathsf{c}) \cdot t \approx ? (b\ \mathsf{c}) \cdot u\} \uplus P$ $(\mathsf{c} \notin \Sigma)$
$\Sigma : \{\pi \cdot X \approx ? t\} \uplus P$	$\longrightarrow$	$\Sigma : P\{\pi^{-1} \cdot t/X\} \uplus \{X \approx ? \pi^{-1} \cdot t\}$ $(X \notin FV(\pi, t))$

Example 7.2.20. Here is a small example of nominal unification:

$$\begin{aligned}
\Sigma : \{f(\langle \mathsf{a}\rangle X, \mathsf{b}) \approx ? f(\langle A\rangle Y, A)\} &\longrightarrow \Sigma : \{\langle \mathsf{a}\rangle X \approx ? \langle A\rangle Y, \mathsf{b} \approx ? A)\} \\
&\longrightarrow \Sigma\#\mathsf{c} : \{(\mathsf{a}\ \mathsf{c}) \cdot X \approx ? (A\ \mathsf{c}) \cdot Y, \mathsf{b} \approx ? A)\} \\
&\longrightarrow \Sigma\#\mathsf{c} : \{(\mathsf{a}\ \mathsf{c}) \cdot X \approx ? (\mathsf{b}\ \mathsf{c}) \cdot Y, A \approx ? \mathsf{b})\} \\
&\longrightarrow \Sigma\#\mathsf{c} : \{X \approx ? (\mathsf{a}\ \mathsf{c}) \cdot (\mathsf{b}\ \mathsf{c}) \cdot Y, A \approx ? \mathsf{b})\}
\end{aligned}$$

Note that this problem reduces to a satisfiable solved form without recourse to name-name unification. This is because the value of A can be determined through ordinary unification steps. Here is a problem for which this is not the case:

$$\begin{aligned}
\Sigma : \{f(\langle B\rangle A, A) \approx ? f(\langle A\rangle B, A)\} &\longrightarrow \Sigma : \{\langle B\rangle A \approx ? \langle A\rangle B, A \approx ? A\} \\
&\longrightarrow \Sigma : \{\langle B\rangle A \approx ? \langle A\rangle B\} \\
&\longrightarrow \Sigma\#\mathsf{c} : \{(B\ \mathsf{c}) \cdot A \approx ? (A\ \mathsf{c}) \cdot B\}
\end{aligned}$$

In this problem, additional work is needed to determine whether the name-name constraint $\{(B\ \mathsf{c}) \cdot A \approx ? (A\ \mathsf{c}) \cdot B\}$ is satisfiable (it is, provided $A = B$).

Lemma 7.2.21. *The following formulas are valid in the Herbrand universe of nominal terms:*

$$\langle\rangle \approx \langle\rangle \tag{7.6}$$
$$t \approx u \iff f(t) \approx f(u) \tag{7.7}$$
$$t_1 \approx u_1 \wedge t_2 \approx u_2 \iff \langle t_1, t_2\rangle \approx \langle u_1, u_2\rangle \tag{7.8}$$
$$\text{И}\mathsf{c}.(a\ \mathsf{c}) \cdot t \approx (b\ \mathsf{c}) \cdot u \iff \langle a\rangle t \approx \langle b\rangle u \tag{7.9}$$

Proof. The forward directions are theorems of NL. The reverse directions of the first three are standard, and the reverse direction of (7.9) is a theorem of NL. □

Theorem 7.2.22. *NU is sound.*

Proof. We must verify that each transition $P \longrightarrow P'$ satisfies $P' \vDash P$. For the first four transitions, soundness follows from the forward directions of (7.6)–(7.9). For the final transition,

$$\Sigma : \{\pi \cdot X \approx ?\ t\} \uplus P \longrightarrow \Sigma : P\{\pi^{-1} \bullet t/X\} \uplus \{X \approx ?\ \pi^{-1} \bullet t\}$$

suppose that $\theta \vDash P\{\pi^{-1} \bullet t/X\}, X \approx \pi^{-1} \bullet t$. Then $\theta \vDash X \approx \pi^{-1} \bullet t$ so $\theta \vDash P$ by substitution. Moreover, $\theta \vDash \pi \cdot X \approx t$ by equivariance. □

Proposition 7.2.23. *NU is terminating.*

Proof. Let $n_1(P)$ be the number of unsolved variables in P. For n_2, let μ be the measure on terms defined in Proposition 7.2.16 that counts the number of units, function symbols, pairings, and abstractions in a term. Set

$$n_2(P) = \sum_{(t \approx ?u) \in P} \mu(t) + \mu(u)\ .$$

Set $\nu(P) = (n_1(P), n_2(P))$.

We claim that each rule of *NU* decreases ν with respect to the lexicographic ordering. For the first four rules, the number of unsolved variables in P remains the same and n_2 decreases. For the last rule, n_1 decreases. □

Proposition 7.2.24. *NU is progressive.*

Proof. Proof is by induction on the size of P and case decomposition of the structure of the constraints within P. Technically we prove that if $\theta \vDash P$ and P is not in solved form then for some Q, $P \longrightarrow Q$ and $\theta \vDash Q$. Let P be given. If P is empty there is nothing to prove. Othereise, $P = \{t \approx ?\ u\} \uplus P'$, and there are several cases:

- If t and u start with the same head symbol, P can take a step by using one of the first four rules. There are several sub-cases to verify, but each case follows from one of the formulas (7.6)–(7.9).

- If both $t = \pi_1 \bullet X$ and $u = \pi_2 \bullet X$, then $t \approx ?\ u$ is in solved form, so we proceed by induction on P'.

- Otherwise, if $t = \pi \cdot X$ and X occurs in π or t, and X is not of name-sort, then $t = \pi \cdot X$ cannot have a solution among finite terms. On the other hand, if X is of name-sort, then both t and u are of the same name-sort and so $t \approx ?\ u$ is a name-name constraint, i.e., in solved form. Therefore, we proceed by induction on P'.

- If $t = \pi \bullet X$ and $X \notin FV(\pi, t)$, then $\theta \vDash X \approx \pi^{-1} \bullet t$. If X appears elsewhere in P', then we can take a variable-elimination step, otherwise X is solved with respect to P, so we proceed by induction.

- Otherwise, neither t nor u is of the form $\pi \cdot X$ and t and u do not start with the same head symbol, so P is unsatisfiable.

This analysis exhausts all cases and completes the proof. $\square$

Corollary 7.2.25. *NU is complete.*

7.2.5 Equivariant Unification

Now we consider equivariant unification problems of the form $p(t) \approx ?\ p(u)$, where p is a relation symbol.

We consider probably the simplest possible (nondeterministic) algorithm for equivariant unification, in which we guess the number of swappings needed for the desired permutation π and then attempt to solve the resulting unification problem. The corresponding constraint solving problem transforms a problem of the form $p(t) \sim ?\ p(u)$ to one of the form $\pi \cdot t \approx ?\ u$, which can then be solved using nominal unification.

The algorithm EVU_0 has two transition rules:

$$\begin{array}{rcl} p(t) \approx ?\ p(u) & \longrightarrow & p((a\ b) \cdot t) \approx ?\ p(u) \\ p(t) \approx ?\ p(u) & \longrightarrow & t \approx u \end{array}$$

Example 7.2.26. Here is a simple example of equivariant unification followed by nominal unification and name-name constraint solving:

$$\begin{array}{rcl} p(\mathsf{a}, \mathsf{b}) \sim ?\ p(\mathsf{b}, Y) & \longrightarrow & p((\mathsf{a}\ \mathsf{b}) \cdot \mathsf{a}, (\mathsf{a}\ \mathsf{b}) \cdot \mathsf{b}) \sim ?\ p(\mathsf{b}, Y) \\ & \longrightarrow & \langle (\mathsf{a}\ \mathsf{b}) \cdot \mathsf{a}, (\mathsf{a}\ \mathsf{b}) \cdot \mathsf{b} \rangle \sim ?\ \langle \mathsf{b}, Y \rangle \\ & \longrightarrow & (\mathsf{a}\ \mathsf{b}) \cdot \mathsf{a} \approx ?\ \mathsf{b}, (\mathsf{a}\ \mathsf{b}) \cdot \mathsf{b} \approx ?\ Y \\ & \longrightarrow & \mathsf{b} \approx ?\ \mathsf{b}, \mathsf{a} \approx ?\ Y \\ & \longrightarrow & Y \approx ?\ \mathsf{a} \end{array}$$

Note that this solution is not unique: $Y \approx \mathsf{c}, Y \approx \mathsf{d}, \ldots$ are also solutions, but $Y \approx \mathsf{b}$ is not a solution since $p(\mathsf{a}, \mathsf{b}) \not\sim ?\ p(\mathsf{b}, \mathsf{b})$.

Theorem 7.2.27. *EVU_0 is sound.*

Proof. Obviously $t \approx u \vDash p(t) \approx p(u)$. By equivariance, $p((a\ b) \cdot t) \approx p(u) \vDash p(t) \approx p(u)$. $\square$

Clearly EVU_0 is nonterminating, so our usual approach to proving completeness will not work. Fortunately, in the Herbrand universe we have $p(\vec{t}) \approx p(\vec{u})$ satisfiable if and only if there is some (finite) permutation π making $\pi \cdot t \approx u$ satisfiable.

Theorem 7.2.28. *EVU_0 is complete.*

Proof. We must show that $P \vDash \bigvee Solv(P)$, where $P = p(t) \approx? p(u)$. Let $\theta \vDash P$ be given. Then $\pi \cdot \theta(t) = \theta(u)$ for some permutation $\pi \in \mathbb{G}$. Let $\tau_1 \cdots \tau_n$ be a transposition representation of π. Then

$$\pi \cdot \theta(t) = \tau_1 \cdots \tau_n \cdot \theta(t) = \theta(\tau_1 \cdots \tau_n \cdot t) \ .$$

Therefore, $\theta \vDash \tau_1 \cdots \tau_n \cdot t \approx? u$. Note that

$$\begin{array}{rcl} p(t) \approx? p(u) & \longrightarrow & p(\tau_n \cdot t) \approx? p(u) \\ & \longrightarrow^* & p(\tau_1 \cdots \tau_n \cdot t) \approx? p(u) \\ & \longrightarrow & \tau_1 \cdots \tau_n \cdot t \approx? u \ . \end{array}$$

so $\tau_1 \cdots \tau_n \cdot t \approx u$ is a solved form of P in EVU_0, so $\theta \vDash \bigvee Solv(P)$; since θ was arbitrary, $P \vDash \bigvee Solv(P)$. □

7.3 Putting It All Together

We have now described all of the ingredients for an idealized, *complete* implementation of Horn clause nominal logic programming. That is, using the semantics for nominal logic programs outlined in the previous chapter, together with the algorithms for name-name and freshness constraint solving and nominal and equivariant unification, we can devise a nondeterministic interpreter NLP for nominal logic programs such that

Theorem 7.3.1 (Soundness and Completeness of NLP). *Let $\mathcal{P}$ be a nominal logic program and G a goal. Let $Solv(G)$ be the set of all solutions obtained by reducing G to constraints C and then solving the constraints using the algorithms of this chapter. Then $\mathcal{P} \vDash G \iff \bigvee Solv(G)$.*

The solved forms for NLP are sets of constraints such that all name-name constraints are solved as in NNU, and all other constraints are solved as in NU and FCS. We omit the details of the construction of the combined logic programming and constraint solving system NLP.

Though theoretically reassuring, there are many practical problems with such an interpreter which would have to be resolved to obtain a usable implementation. The most serious problems are the **NP**-completeness and nondeterminism of name-name constraint solving, and the nontermination of the naïve equivariant unification algorithm. In the rest of this section, we consider ways to address these problems in a practical implementation. We also consider applications of Urban et al.'s nominal unification algorithm, which only works for a restricted sublanguage of full nominal logic programs but produces most general unifiers, yet can be implemented efficiently.

7.3.1 Name-Name Constraints

Because name-name constraint solving is **NP**-complete, determining whether a particular set of constraints is satisfiable may require exponential time. More seriously, our algorithm may produce exponentially many solutions to each constraint, leading to "thrashing" behavior due to backtracking.

In practice, programmers usually use high-level programming features to solve hard problems, rather than encoding them as constraints. For example, many logic programs can be encoded as higher-order unification problems [79], but few programmers write programs this way. Therefore, the constraints encountered in practice are often easier to solve than the worst-case complexity of general constraint solving would suggest.

A popular approach to working with complex constraints and nondeterminism in other constraint logic programming languages is to identify certain constraints as being "hard" and delaying the solution of hard constraints as long as possible, in the hope that they will become easier when more information is available.

In NLP, name-name constraints seem an obvious candidate for "hard" constraints. In addition, simplification rules could be applied to simplify some "easy" name-name constraints. For example, variable elimination can sometimes be applied eagerly, and swapping rules such as $(a\ b) \cdot a \approx b$ can always be applied eagerly.

More generally, it would be desirable to find a formulation of name-name constraint solving that produces unique most general solutions, or to establish that no such formulation exists.

7.3.2 Equivariant Unification

As with name-name constraints, equivariant unification is **NP**-complete, and the algorithm we presented may find infinitely many answers.

One way to deal with the nontermination problem is to modify the interpreter to explore the search space in a deterministic but "fair" way, so that every answer will eventually be found. One fair search strategy is *iterative deepening*: that is, first all answers requiring one reduction step are produced then all answers requiring two steps, etc.

However, this seems like overkill. Equivariance problems (at least in practice) do not seem to require infinitely many different solutions; we conjecture that equivariant unification is in **NP** and that complete finite sets of solutions exist.

As formulated in Section 7.2.5, equivariance problems cannot be delayed as conveniently as name-name constraints, since they may involve first-order structure. Instead, it may be worthwhile to consider alternative approaches to solving equivariant unification problems. We consider two possibilities.

- Add a new similarity relation $\sim$ on ground terms such that $t \sim u$ iff for some permutation π, $\pi \cdot t \approx u$. Then $p(t) \approx p(u)$ iff $t \sim u$.

Now consider similarity problems of the form $\vec{t} \sim ?\ \vec{u}$, with rewriting rules such as

$$\begin{array}{rcl}
\Sigma : \langle X, \vec{t}\rangle \sim ?\ \langle f(u), \vec{u'}\rangle & \longrightarrow & \Sigma, X' : \{\langle X', \vec{t}\rangle \sim ?\ \langle u, \vec{u'}\rangle, \\
& & \quad X \approx ?\ f(X')\} \\
\Sigma : \langle X, \vec{t}\rangle \sim ?\ \langle\langle a\rangle u, \vec{u'}\rangle & \longrightarrow & \Sigma \# \mathsf{c}, X' : \{X \approx ?\ \langle \mathsf{c}\rangle X', \\
& & \quad \langle \mathsf{c}, X', \vec{t}\rangle \sim ?\ \langle \mathsf{c}, (a\ \mathsf{c}) \cdot u, \vec{u'}\rangle\} \\
\Sigma : \langle \mathsf{a}, \mathsf{a}, \vec{t}\rangle \sim ?\ \langle u_1, u_2, \vec{u}\rangle & \longrightarrow & \Sigma : \langle \mathsf{a}, \vec{t}\rangle \sim ?\ \langle u_1, \vec{u}\rangle, u_1 \approx ?\ u_2
\end{array}$$

This approach seems promising and we have performed some preliminary experiments with it, but getting the reduction rules and solved forms right has proven tricky.

- Extend the term language to include the full language of (permutation) group theory (including permutation variables) for permutations π in actions $\pi \cdot t$. Then equivariant unification $p(t) \approx ?\ p(u)$ reduces to solving the satisfiability problem $P \cdot t \approx u$, where P is a permutation variable. Like swappings, general permutations can be pushed down to the lowest level of a term, so equivariant unification problems can be reduced to name-name constraints involving arbitrary permutations.

  This approach is appealing since it reduces equivariant unification to equational unification in a richer (but well understood) theory, that is, group theory. Though unification in group theory has been studied [4], there appears to be no work on unifying equations involving permutation groups or actions. A significant potential problem with this approach is that it seems likely to produce very "fuzzy" answers $X = P_1 \cdot P_2 \cdot P_3 \cdot \mathsf{a}$ involving many permutation variables, one for each backchaining step. Concrete answers involving specific names seem preferable.

We believe a careful analysis of one of the above approaches could be used to show that equivariant unification is in **NP**.

7.3.3 Early Failure Detection

Another problem is detecting failure. In the algorithms so far no attempt is made to stop early when a failure occurs, that is, when the constraint becomes unsatisfiable, and unsatisfiable answers may be produced (e.g. $X \# \mathsf{a} : X \approx \mathsf{a}$). These are disadvantages for a practical implementation, leading to wasted effort and inefficiency. It is desirable to avoid these problems by performing basic satisfiability checking so that failure and backtracking can occur as soon as possible.

7.3.4 Efficient Special Cases

Urban, Pitts, and Gabbay [126, 127] developed a unification algorithm for a special case of nominal terms, which we call *nominal patterns*. Nominal patterns are constraints in which for every term or constraint of the form $(a\ b) \cdot t$, $\langle a \rangle t$, or $a \mathrel{\#} t$, we have a, b ground name-constants. Thus, $\langle A \rangle t$ and $(X\ (Y\ Z) \cdot t) \cdot u$ are not nominal patterns. To distinguish their algorithm from our algorithms for general nominal constraint solving, we refer to their algorithm as *nominal pattern unification*. This terminology is inspired by that used for the *higher-order pattern* sublanguage of higher-order terms for which higher-order unification is efficiently decidable.

Nominal pattern unification produces unique most general unifiers and can be implemented in $O(n^2)$ time [127]. Therefore, it is possible to implement an efficient logic programming language based on nominal patterns. The nominal terms used in αProlog, as defined in Chapter 2, are actually nominal patterns, and nominal pattern unification is used instead of full nominal and equivariant unification.

This results in an efficient, but not necessarily logically sensible programming language. Nominal pattern unification solves only equational and freshness constraints, not equivariance constraints. Therefore, for many αProlog programs and goals, there are answers that are correct with respect to nominal logic but cannot be derived by nominal pattern unification-based backchaining. Probably the simplest possible example is the program $p(\mathsf{a})$ and goal $p(\mathsf{b})$ which we discussed in Chapter 2.

Perhaps surprisingly, however, many typical programs do not suffer from this problem. In particular, programs that deal with bound names in a structured way can be run using nominal pattern unification without missing any answers. For example, the typechecking, substitution, and evaluation programs in Section 2.2.1 can be implemented correctly using nominal patterns alone. Other programs, such as α-inequivalence and closure-conversion, seem to require equivariance to function correctly.

The reason that many programs run correctly without equivariant unification is that equivariance can often be simulated via substitution. Therefore, equivariance can be avoided by avoiding "free" names in a term. As a simple example, the program $p(A)$ defines the same relation as $p(\mathsf{a})$, since $\text{И}\mathsf{a}.p(\mathsf{a}) \iff \forall A.p(A)$ holds in NL. A more complex example is a program like *typ* for typechecking the λ-calculus: there is no essential dependence on the name x in the clause

$$typ(G, lam(\langle \mathsf{x} \rangle M), arr(T,U)) \mathrel{:-} \mathsf{x} \mathrel{\#} G, typ([(\mathsf{x}, T)|G], M, U).$$

It is possible to show that (for a slight modification of this clause) any backchaining step taken using equivariant unification can be simulated using nominal unification. This proof can be generalized to identify a class of programs for which nominal pattern unification-based resolution is complete. These programs form a restricted nominal logic programming language that is both logically sensible and efficiently implementable.

The above discussion is a highly condensed version of work in progress by Urban and Cheney [125].

7.4 Notes

The complexity results for equivariant unification and matching reported in Section 7.1 first appeared in [19]. Goldmann and Russell [47] showed that for finite groups, solving systems of equations (possibly involving constants) is in **P** if the group is Abelian, otherwise **NP**-complete. This result inspired our first workable proof of **NP**-hardness for equivariant unification.

Equational unification, that is, unification modulo an equational theory E (also known as E-unification) has been studied by many authors. Some references that we have relied upon include Baader and Nipkow's book [9] and Snyder's monograph [119]. General E-unification techniques, such as narrowing [118, 34], impose restrictions (for example, confluence) on E in order to guarantee completeness; however, equivariance in Nominal Logic obviously breaks confluence. Snyder's algorithm, and later variants such as Lynch's goal-directed E-unification [71], are exceptions in that they work for any equational theory, as opposed to just confluent theories. It would be interesting future work to determine whether any extant E-unification algorithms can be used to obtain a terminating algorithm for equivariant unification. However, the structure of nominal and equivariant unification problems (for example, the fact that swappings can always be pushed down past abstractions and function symbols) seems to make our more direct approach preferable.

Chapter 8

Related Work

Imitation is the sincerest form of flattery.

—Charles Caleb Colton

The purpose of this chapter is to compare the techniques proposed in this thesis with other proposed techniques for programming with names and binding. Because of the great importance of this problem in programming and in formal reasoning, there is a vast body of related work in the areas of logic, functional and logic programming language design, automated reasoning, logical frameworks, and the theory of abstract syntax. It is impossible to give a complete, yet concise survey of all the relevant work.

Instead, we will only compare nominal logic programming with the most closely related approaches. In particular, we will focus on programming languages providing advanced support for *programming with* names and binding and not on abstract syntax encoding techniques for *reasoning about* names and binding in a logical framework or theorem prover. The two tasks, and the underlying techniques, are related, but the emphasis of this dissertation has been on programming techniques, not on automated reasoning and logical frameworks based on nominal abstract syntax.

We classify existing approaches to programming with names and binding by the technique used to deal with binding.

- Name-free approaches: combinatory logic [112, 25, 26], de Bruijn indices [30]
- Higher-order abstract syntax: λProlog [90], Twelf [105], Delphin [115]
- Lambda-term abstract syntax: L_λ [78], ML_λ [77].
- A first-order theory of α-equivalence: Qu-Prolog [121, 95]
- Binding algebras: Hamana [48]
- Nominal abstract syntax (FreshML) [116, 117, 109]

Each of these approaches is discussed and compared with αProlog in Section 8.1. In Section 8.2, we discuss some additional relevant research.

8.1 Programming Languages and Techniques

8.1.1 Name-Free Approaches

In addition to the first-order and higher-order approaches, there are several approaches to dealing with names and binding by getting rid of names entirely while retaining the expressiveness of the λ-calculus. These approaches have a long history, beginning with Schönfinkel's development of combinatory logic in 1920-24 [113], and have had considerable influence on both theory and practice of logic and programming.

Combinatory Logic

Schönfinkel [113] and later Curry and Feys [25, 26] developed *combinatory logic*, a logic of applicative expressions defined using rewriting rules. A combinatory logic expression e is either a constant A, B, C or an application $(e\ e')$, where we write $e\ e_1\ \cdots\ e_n$ for $(\cdots(e\ e_1)\cdots e_n)$ to omit parentheses when no ambiguity ensues. For example, the S, K and I-combinators are defined by equations

$$Ix = x \qquad Kxy = x \qquad Sxyz = (xz)(yz)$$

A term to which none of these equations can be applied (in left-to-right order) is in *normal form.* A term e can therefore be interpreted as a partial function $x \mapsto e\ x$ on normal forms, computed by attempting to normalize $e\ x$.

There is an obvious translation from combinator terms to simply typed λ-terms to over S, K, I: take $I = \lambda x.x$, $K = \lambda x, y.x$, and $S = \lambda x, y, z.(xz)(yz)$. Conversely, there is a more involved translation from the simply-typed λ-calculus to combinator terms. So in theory, one could work exclusively in terms of combinators and avoid the issue of naming entirely. However, in practice this is not feasible, since the translation from λ-terms to combinators may increase the size of terms exponentially. In addition, combinator expressions are much less readable than their equivalent λ-calculus versions. For example, the composition combinator Z is $S(KS)K$ in combinatory form, but $\lambda f.\lambda g.\lambda x.f(g(x))$ in the λ-calculus. On the other hand, an extension of the combinator approach called *categorical combinators* (drawing on ideas from category theory) leads to a practical implementation technique for functional programming languages [24].

De Bruijn Indices and Levels

N. G. de Bruijn [30] proposed two encodings for the λ-calculus which neatly circumvent the difficulties arising from α-equivalence. In both encodings, the following grammar is used for λ-terms:

$$e ::= \lambda e \mid e\ e' \mid n$$

where $n \in \{1, \ldots\}$. Both encodings replace textual occurrences of variables with numerical references; the encodings differ only in how the numbers are interpreted.

In the *de Bruijn indices* encoding, n refers to the nth enclosing λ, counting out from n. For example, we have the following encodings of ordinary λ-terms:

$$\ulcorner \lambda x.\lambda y.(x\ y)\urcorner = \lambda\lambda(2\ 1) \qquad \ulcorner \lambda x.x\ (\lambda y.(y\ x))\urcorner = \lambda(1\ (\lambda(1\ 2)))$$

In the *de Bruijn levels* encoding, n refers to the nth λ on the path from the root of the term to the occurrence of n, counting from the top level of the term. The same two terms are represented as follows:

$$\ulcorner \lambda x.\lambda y.(x\ y)\urcorner = \lambda\lambda(1\ 2) \qquad \ulcorner \lambda x.x\ (\lambda y.(y\ x))\urcorner = \lambda(1\ (\lambda(2\ 1)))$$

The two encodings share the advantage that α-equivalent closed λ-terms have syntactically equal de Bruijn encodings. On the other hand, both share the disadvantage of being practically unreadable by humans, and neither provides support for programming with open terms. In addition, while α-equivalence becomes trivial, the substitution and β-reduction operations may be relatively complex since variable references may need to be renumbered when the binding structure of the terms changes or when a term is substituted under a λ. In particular,

$$(\lambda t)\{t'/n\} = \lambda(t\{\uparrow t'/n+1\})$$

where $\uparrow t'$ is the result of incrementing every free variable number in t by one. In contrast, the λ-case for substitution in αProlog is essentially the same as the one used in informal presentations.

Nevertheless, de Bruijn indices are the basis of many efficient implementations of higher-order functional and logic programming languages, logical frameworks, and theorem proving systems, as an efficient internal representation for λ-terms. The behavior of substitution in the de Bruijn indices encoding is well-understood: explicit substitution calculi such as $\lambda\sigma$ have been developed in which substitutions are treated as first-class terms subject to rewriting rules [1], and efficient implementations of higher-order (pattern) unification have been developed for $\lambda\sigma$ [33]. de Bruijn representations have also been investigated using functor-category semantics [37, 54].

8.1.2 Higher-Order Abstract Syntax

Higher-order abstract syntax [104], as mentioned in the Introduction, is the technique of defining all binders in terms of λ: that is, the λ-abstraction provided by the meta-language is used to implement all variable binding constructs in the object languages of interest. It was first employed by Church, who defined the quantifiers of higher-order logic as constants of types like $\forall_\tau : (\tau \to \mathbf{prop}) \to \mathbf{prop}$.

The logic programming language λProlog [90] is based directly on higher-order logic and employs Church's technique for dealing with the quantifiers directly. Twelf [105] is a logic programming language based on the Edinburgh Logical Framework, a dependently-typed lambda calculus (called $\lambda\Pi$ or LF). Delphin [115] is a functional programming language, currently under development, incorporating dependent typing and other ideas from logical frameworks, Twelf in particular. Twelf and Delphin follow the *propositions as types* principle, and so can produce *proof terms* explicating the derivation of any answer upon request. This makes both languages suitable for both programming (searching for proofs) and proof-checking.

All of these languages are extremely powerful and expressive. Twelf, in particular, can be used to describe programming languages, reductions, and type systems and also to prove important properties thereof, and this is also an aim of Delphin. But they are also limited in some ways because of their dependence on higher-order abstract syntax. As a result, there are very simple programs that can be written for first-order abstract syntax or nominal abstract syntax but not written as easily for higher-order abstract syntax. Here are several examples:

- Consider the relation *neq* that takes two object-level λ-terms and checks that they are *not* equal up to α-equivalence, defined in Example 2.2.3. This program cannot be written cleanly in a higher-order encoding, because it is meaningless to compare the syntactic names of the bound variables of a term since they only represent other values, and are not values themselves. Instead it is necessary to do some additional processing, such as tagging variables with numbers that can be compared.

  In αProlog, names are themselves concrete values and can be tested for both equality and inequality (freshness). For the same reason, the closure-conversion program of Example 2.2.5 cannot be written using higher-order abstract syntax without additional effort.

- It is difficult to simulate imperative features such as references in such languages, because the changing state of a reference cell cannot be modeled in the hypothesis context. This motivated the development of Linear LF [15], an extension of LF with linear logic primitives permitting hypotheses to be added, deleted, and changed over time.

  In αProlog, we take a quite different approach: the reference store is not implicitly encoded as an augmentation of the program, but an explicit list of name-value pairs. This approach, while possible in any language, would be significantly more complicated in an ordinary programming language because of the lack of support for name-generation to generate fresh reference cell identifiers.

- It is difficult to model a concurrent language such as the π-calculus in Twelf or LLF, because the nondeterminism of truly concurrent programs cannot be expressed properly in LLF. This motivated the development of Concurrent LF, or CLF, an extension of LLF with a monadic type and asynchronous primitives from linear logic [130, 131].

> In αProlog, our sights are set much lower than in CLF. We can implement the π-calculus transition system and simulation/bisimulation relations directly and more-or-less as originally designed, so there can be little doubt that they are correct (in the sense of behaving as specified on paper). We are not concerned with automated reasoning about these relations, but there seems to be no obstacle to doing so within ordinary first-order nominal logic.

In addition, the high semantic and algorithmic complexity of higher-order abstract syntax means that implementing interpreters, compilers, and program analyses for these languages can be difficult. Whereas unification of higher-order terms is undecidable, equivariant unification is only **NP**-hard and, we believe, in **NP**. Al αProlog is a relatively minor extension of ordinary (typed) first-order constraint logic programming, there is some hope that existing mode and termination analyses can be generalized to αProlog without much difficulty.

In favor of λProlog, Twelf, Delphin, and other higher-order abstract syntax-based languages is the fact that substitution and α-equivalence only needs to be implemented once, by the language implementor, rather than over and over by the programmer. In αProlog, substitution needs to be implemented explicitly, but there is some hope of automating this process. In fact, we have already experimented with a built-in capture-avoiding substitution operator in the language implementation, along the lines of the substitution operation of Section 3.5.5.

Another positive aspect of Twelf and its descendants is the high degree of elegance of its treatment of logics, programming languages with state (LLF), and now, concurrency (CLF). This elegance is a strong and often-cited argument in favor of the overall LF approach, yet higher-order encodings are often very different from the intuitive formulations most often found in books and research papers. In nominal abstract syntax, on the other hand, "paper" mathematical specifications can usually be converted directly into nominal logic programs, leaving little doubt as to the correctness of the translation. This fact suggests that nominal abstract syntax, while not elegant *in the same way* as higher-order abstract syntax, nevertheless possesses its own elegance. Of course, beauty is in the eye of the beholder.

8.1.3 Lambda-Term Abstract Syntax

A significant complication for higher-order techniques is that higher-order unification (which is needed for backchaining in higher-order logic) is undecidable [58, 59]. In addition, most general unifiers may not exist or may not be unique when they do exist. However, in practice many programs do not seem to encounter these problems, and Huet's technique of pre-unification (unification with "hard" subproblems delayed) is effective in most practical situations. To explain and take advantage of this insight, Miller [78] identified a decidable and well-behaved special case of higher-order unification called *higher-order pattern unification.* A higher-order pattern is a λ-term in which each meta-variable (that is, logical variable for which a term may be substituted) is applied to a list of *distinct* bound variables. For example, if F, G are meta-variables, then $\lambda x, y.F\ x\ y$ is a higher-order pattern, but $\lambda x.F\ x\ x$ and $\lambda x.F\ G\ x$ are not patterns. Miller discovered that all of the nondeterminism in Huet's higher-order pre-unification algorithm can be avoided if the unificands are patterns, and most general unifiers are unique and can be computed efficiently (in fact, in linear time, as shown by Qian [111]). Moreover, the equational theory of higher-order patterns can be viewed as a special case of full $\alpha\beta\eta$-equivalence, in which full β-reduction

$$(\lambda x.t)\ u \longrightarrow_{\beta} t[u/x]$$

is replaced with "β_0-reduction"

$$(\lambda x.t)\ y \longrightarrow_{\beta_0} t[y/x]$$

that is, the left-hand side of an application is required to be a variable.

Lambda-term abstract syntax is a refinement of higher-order abstract syntax introduced by Miller [80], in which the λ-terms are restricted to be higher-order patterns. Miller observed that many interesting λProlog programs can be written using only patterns, and proposed a logic programming language L_λ, essentially the subset of λProlog formed by restricting to higher-order patterns. That is, in full λProlog, the beta-reduction predicate can be encoded as

```
beta (app (lam (x\E x)) E') (E E').
```

but this is not a higher-order pattern because of the subterm `E E'`. 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 addition, Miller 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 [77]. 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.

There may be important connections between higher-order patterns and nominal patterns, and between λ-term abstract syntax and nominal abstract syntax.

In both, general substitution is eschewed in favor of replacing variables with variables or swapping names with names. In LTAS, the dependence of meta-variables on names is given positively (that is, writing $F\ x$ means F can depend only on x); in NAS, it is negative (that is, writing $x \# F$ means F can depend on any name except for x). In both higher-order pattern unification and nominal pattern unification, permutations are used in the case of unifying two occurrences of the same logical variable. It is possible that nominal pattern unification can be implemented using higher-order pattern unification or vice versa. One direction of this problem is considered by Urban et al. [127]. It appears that nominal unification can be decided by reducing the problem to higher-order pattern unification, but it is difficult to convert the resulting higher-order unifier to a nominal unifier; in particular freshness constraints are difficult to recover. The reverse direction has not yet been studied.

Lambda-term abstract syntax shares some of the difficulties of HOAS. NAS retains the advantage that names are concrete data and there is no problem dealing with open terms or name inequality. Lambda-term abstract syntax, in contrast, cannot handle these issues any better than higher-order abstract syntax techniques can. It is also not clear how much simpler LTAS is than full HOAS in semantic terms, since the denotational semantics of LTAS does not appear to have been investigated (but see the next section for further discussion). In contrast, nominal abstract syntax has a solid semantic foundation using nominal sets. It is possible that a semantics of LTAS could be found in terms of nominal sets as well.

On the other hand, efficient (linear time) algorithms for higher-order pattern unification are well-known, whereas equivariant unification is **NP**-hard and tight bounds on the complexity of nominal pattern unification has not been established.

8.1.4 Binding Algebras

Fiore, Plotkin, and Turi [37] developed a new semantic approach to abstract syntax with binding based on category theory. They consider the functor category $Set^{\mathbb{F}}$, or intuitively, sets acted upon by functions among finite sets of names (i.e., renamings). They showed how to construct an object δA from an object A of this category: essentially, δA is an A with a single bound variable. They showed that δA can be used, in conjunction with pairing and induction, to construct models of abstract syntax with variable binding equivalent to de Bruijn encodings. (In fact, they showed explicitly how de Bruijn indices and de Bruijn levels could be obtained by varying the details of the construction.) They also considered refinements of such categories with a built-in notion of substitution. See also Hofmann [54] for a similar development.

Hamana [48] has developed a unification algorithm and logic programming language for programming with 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. These terms are similar to nominal patterns and also to higher-order patterns. For example, $\langle \mathsf{a} \rangle \langle \mathsf{b} \rangle (\mathsf{a}\ \mathsf{b}) \cdot X$,

$\lambda a, b.X\ b\ a$, and $[a][b][a := b, b := a]X$, all play roughly the same role in NAS, LTAS, and Hamana's language, respectively.

Hamana's unification algorithm unifies up to β_0-equivalence of bound names with respect to name-application. That is, Hamana's terms are unified subject to the β_0 equation $([a]t)@b = [a := b]t$, as well as α-equivalence $[a]t = [b]u \iff [a := c]t = [b := c]u$ for some "new" c. In order to ensure that a "new" name can be chosen, Hamana employs a type system that assigns each term a type plus the set of names that may appear free in the term. Thus, the new c in α-equivalence must be chosen not already in the sets supporting t and u, so the α-equivalence rule really looks like:

$$S \lhd [a]t = [b]u \iff S, c \lhd [a := c]t = [b := c]u \quad (c \notin S)$$

(This is not the actual α-rule used by Hamana, but gives the basic idea.) Because Hamana's unification algorithm unifies up to β_0-equivalence without requiring terms to be higher-order patterns, it is a generalization of higher-order pattern unification. In addition, logical variables may contain free names. However, this is not as big a difference as it might seem, since the free names that may appear in a term are specified as part of its type. Therefore, the lifting of the higher-order pattern restriction appears to be the only substantial difference between Hamana's algorithm and higher-order pattern unification. Because it unifies terms that may not obey the pattern restriction, Hamana's algorithm can produce multiple incomparable most general unifiers. Its complexity has not been investigated, but it seems likely to be at least **NP**-hard, since it solves a *unification up to injective renaming* problem that is similar to equivariant unification. It is therefore of interest to determine whether Hamana's algorithm reduces to equivariant unification problems or vice versa.

Many of the example programs of Chapter 2 can also be programmed using Hamana's programming language. For example, capture-avoiding substitution is given as an example in Hamana [48]. However, this version is more limited than the αProlog version because Hamana's language requires the free names that may appear in a term to be declared explicitly as part of its type. For this reason, it is apparently not possible to substitute an open term for a variable using Hamana's version, since the substitutend must be closed. (This is necessary to ensure that substitution is capture-avoiding.) More generally, Hamana's language appears to have the same problems with open terms as higher-order abstract syntax, so examples like α-inequivalence and closure conversion would not be easy to write. This is not surprising since Fiore et al.'s work on binding algebras takes no account of open terms, only terms in some finite variable context.

Hamana's language has not been implemented, and the issues of typechecking and especially type inference that accompany its type system have not been studied, as far as we know. Therefore, it is not possible to compare Hamana's language with αProlog directly. However, it is hard to see how Hamana's type system could be any simpler than that of αProlog, and the fact that name-supports appear in types seems likely to be a significant complication.

On the other hand, Hamana's language rests on the solid semantic foundations of Fiore et al's semantics for binding algebras and there is no theoretical obstacle to defining functions by iteration over binding algebra terms or to performing inductive reasoning. Therefore Hamana's language is similar to αProlog in theoretical transparency. The apparently close relationship between Hamana's language and λ-term abstract syntax also bears examination: Hamana's techniques may indicate that LTAS can be given a semantics in terms of binding algebras. Hofmann's semantic analysis of higher-order abstract syntax [54] also seems to bear this out.

8.1.5 Qu-Prolog

Qu-Prolog [121, 95] 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 variables and a built-in capture-avoiding substitution operation $t\{t'/x\}$. 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. As in binding algebras, and unlike in HOAS, 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. Unification of quantifiers proceeds by first substituting the bound names with a fresh name. It is therefore necessary for Qu-Prolog unification to track "not-free-in" constraints as well as to deal with capture-avoiding substitution as a built-in function symbol. In addition, capture-avoiding substitution is useful in its own right. However, Qu-Prolog unification is undecidable in general, so in practice full unification is not performed; instead, unification problems are reduced to constraints which may involve not_free_in or irreducible substitution problems [95]. This approach is reminiscent of Huet's pre-unification algorithm for nominal terms.

In several respects Qu-Prolog is very similar to αProlog. Binding is treated in a first-order manner using substitution instead of swapping, and unbound object-variables may appear in terms, but may also be unified with other object-variables. And in some important respects αProlog (at least as developed in this dissertation) is weaker: there is no support for capture-avoiding substitution in αProlog. On the other hand, unification in αProlog is decidable. Many of the same kinds of programs can be written in Qu-Prolog and αProlog. In particular, programs like the λ-calculus reduction and typechecking examples, as well as interpreters and theorem provers, can be and have been implemented in Qu-Prolog relying on its support for names and binding.

However, the semantics or logical foundations of Qu-Prolog do not appear to have been studied. Therefore, it is not clear just what is computed by a Qu-Prolog program involving names and binding. And it is not clear how to read Qu-Prolog programs as logical specifications because the logic is left implicit. This seems like an especially important issue when dealing with names and binding, whose nature is somewhat mysterious. And object-variables have somewhat strange behavior: syntactically distinct names are sometimes assumed to be different values but sometimes may be unified. There is no notion of equivariance nor is there a logical explanation of fresh name generation as is provided by the И-quantifier.

As a consequence of the fact that object-variables can be unified, Qu-Prolog must conservatively produce answers that are more complex than would be necessary in αProlog. For example even if a seemingly correct answer is supplied, Qu-Prolog lacks enough information to be able to solve all the remaining constraints.

```
| ?- [x1/x](lambda x1 x) = lambda x2 x1.
x1 = x1, x = x, x2 = x2
provided:
x1 = [x1/x, x2/x1]x, x2 not_free_in [x1/x, $/x1]x
```

In αProlog, a similar query such as

$$?\!\!- \ subst(lam(\langle \mathsf{x}_1\rangle var(\mathsf{x})), var(\mathsf{x}_1), \mathsf{x}, lam(\langle \mathsf{x}_2\rangle (\mathsf{x}_1)))$$

succeeds unconditionally. Of course, the NLP query

$$?\!\!- \ subst(lam(\langle A_1\rangle var(A)), var(A_1), A, lam(\langle A\rangle var(A_1)))$$

would reduce to some constraints similar to the above, but this query is not allowed in αProlog because it violates the pattern restriction. Thus, the presence of name-constants rather than just name-variables (or object-vars) is a real difference between αProlog and Qu-Prolog.

Nevertheless, many interesting programs can be written in Qu-Prolog, including interactive theorem provers, client/server and database applications, and the Qu-Prolog interpreter and compiler themselves. For comparison with other techniques for programming with names examined so far, here is a λ-term typability relation expressed in Qu-Prolog notation.

```
type(A(B), Y, TypeAssign) :-
        !, type(A, X ~> Y, TypeAssign), type(B, X, TypeAssign).
type(lambda x A, T ~> TA, TypeAssign) :-
        !, type(A, TA, [x^T|TypeAssign]).
type(X, TX, TypeAssign) :-
        in_type(X^TX, TypeAssign).
```

(This code is copied from the Qu-Prolog distribution). Here, `X^TX` and `T~>U` are user-defined binary operators corresponding to $x : \tau$ and $\tau \to \tau'$ in Qu-Prolog, and `in_type` is a predicate which looks up the *most recent* binding of `X^TX` in the type assignment `TypeAssign`. This use of cut is necessary for correctness. In αProlog, cut is not necessary in this situation.

Qu-Prolog enjoys a very mature implementation including a compiler for Qu-Prolog written in Qu-Prolog. It is possible that Qu-Prolog could be given a semantics in terms of nominal abstract syntax (with a built-in notion of substitution in addition to swapping). Conversely, Qu-Prolog's built-in capture-avoiding substitution operator has inspired experimentation with a similar operator in αProlog.

8.1.6 FreshML

FreshML [109, 116, 117] was an important source of inspiration for αProlog. At present FreshML and αProlog provide almost identical facilities for dealing with nominal abstract syntax itself (that is, name-types, name-abstraction types, and so on). However, in early versions of FreshML, a complicated type analysis was employed to ensure that name-generating functions were "pure" (side-effect-free); this analysis was found to be overly restrictive and has been dropped in recent versions resulting in a language that is more permissive but has side-effects. In contrast, fresh name generation in αProlog leads not to side-effects but to nondeterminism—which is almost as bad as side-effects in a functional language, but which is perfectly acceptable in a logic programming language.

Another difference is that there are no concrete names in FreshML; instead, names are always manipulated via variables. On the other hand, in recent versions of FreshML, names may be constructed from data such as strings and integers; also, data structures containing names may be bound, not just individual names. These seems like useful features that could also be incorporated into αProlog.

The rest of the differences are ordinary differences between functional and logic programming. FreshML is a higher-order programming language with first-class functions, whereas αProlog is limited to first-order programming, and functions are not first-class. From a theoretical point of view, there is no apparent obstacle to adding first-class functions to αProlog. It would also be interesting to generalize αProlog to allow genuine higher-order logic programming as in λProlog. This would be a nontrivial project; a first step in this direction would be investigating the combined nominal-higher-order unification problem.

Conversely, there are many programs that can be written cleanly in αProlog but not so cleanly in FreshML. Typechecking programs are a typical example, because they often require simultaneous checking and inferring of types, which is easily handled in a logic programming language using unification. In a functional language, it is usually necessary to write two mutually recursive functions, one for type-checking and one for type-inference, whereas in a logic programming language both functions can be performed by the same set of clauses. (In fact, some implementations of typechecking and inference in functional languages essentially simulate logic variables and unification so that typechecking and inference can be performed by a single function [64]).

8.2 Additional Related Work

8.2.1 Name-Generation in Programming

Pitts and Stark [110] and Odersky [97] studied name-generation in functional programming languages.

Pitts and Stark's nu-calculus analyzed name generation as an effectful computation, analogous to reference generation in ML. Names could be introduced with a "fresh name" binder $\nu n.t$, and tested for equality. The fresh name construction was interpreted operationally by maintaining a name-store: on encountering a $\nu n.t$ term, a fresh name is bound to n and added to the store. Pitts and Stark found that, contrary to expectation, the interaction between higher-order functions and name-generation is very complex. In addition, because name-generation involves side-effects, the nu-calculus is not confluent, and has weaker equational properties than a purely functional language. In some ways, this work can be seen as an early precursor to that of Pitts and Gabbay on FreshML [109].

Odersky developed a quite different functional theory of local names [97]. His $\lambda\nu$-calculus is syntactically essentially the same as Pitts and Stark's nu-calculus. But instead of treating name-generation as an effect, the $\lambda\nu$-calculus deals with names in a local way. As a result, the strong equational properties of the λ-calculus are preserved in the $\lambda\nu$-calculus, which remains a purely functional language. On the other hand, the local names behave strangely in some ways: if local names cannot be removed from a term, then that term is not a value. For example, the term $(\lambda x.x = x)\nu n.n$ reduces to $\nu n.n = \nu n.n$, which is stuck $\nu n.n$ is not a value. In other ways, ν behaves much like the π-calculus restriction operator: for example, $\nu n.M \cong M$ if n is fresh for M, and $\nu n.\nu m.M \cong \nu m.\nu n.M$ (where $\cong$ means observational equivalence). Most interestingly, Odersky developed a denotational semantics for $\lambda\nu$ in terms of name-swapping and support. This development clearly foreshadows the later developments underlying nominal logic, FreshML, and αProlog, although, of course, without the application to variable-binding. The relationship between $\lambda\nu$ and FreshML is therefore of great interest, as is the possibility of using $\lambda\nu$ as a proof-term calculus for nominal logic.

8.2.2 Meta-Programming with Names and Necessity

Nanevski and Pfenning [92] have developed intriguing applications of nominal abstract syntax to meta-programming. They have combined explicit concrete names with a modal type system (based on a proof-term calculus $\lambda^{\Box}$ for intuitionistic S4 modal logic developed by Davies and Pfenning [29]). This combination yields a powerful language for meta-programming via staged computation, that is, writing programs that write programs that can be compiled and run at run-time.

αProlog itself is a powerful meta-programming language, since the abstract syntax of a language can be represented cleanly using αProlog terms and then programs can be constructed by αProlog programs. But such programs must be converted to some other format before they can be run; being a typed language, αProlog does not even allow meta-circular interpretation of αProlog programs. So αProlog can be viewed as a meta-programming language in only a fairly weak sense.

In contrast, $\lambda^\Box$ and $\nu^\Box$ are among the most recent in a line of languages for writing ML-like programs that construct ML-like programs that can be compiled at run time. The role of the "box" modality $\Box$ in $\lambda^\Box$ is to separate the early stage of computation that is manipulating some code from the late stage that is being manipulated. However, in order to achieve type safety, $\lambda^\Box$ is very conservative about communication between the levels: in fact, boxed terms containing code generated at run time always have to be closed, to avoid prevent variables from being evaluated before values has been assigned to them. This problem is called "free variable evaluation", and is solved in some other systems by dynamic typechecking [122] or a complicated type system [16]. This problem was solved in $\lambda^\Box$ by prohibiting free variables in boxed expressions. This is safe but overly restrictive and forces programs to be written in a contorted style.

The $\nu^\Box$ language solves the free variable evaluation problem by tracking free variables (or "support") of a boxed term. A boxed term can only be "run" (or compiled to executable code) if it has empty support, that is, mentions no free ν-variables. (Ordinary variables already are always guaranteed to be assigned values during the execution of the current stage). ν-variables can be removed from the support of a term by applying an *explicit substitution*, that associates a name with a term. Of course, substitutions can introduce names as well as eliminate them, so in general the source and target name-sets of substitutions appear in the type system as well.

While $\nu^\Box$ is very interesting, the most recent version of $\nu^\Box$ seems only tangentially related to nominal logic. Nanevski's original $\nu^\Box$ system [91] was based very closely on ideas from nominal logic, including swapping and freshness. In the current version, the only connections are the use of explicit names as data which are syntactically distinguished from λ-bound variables and the idea of "support" tracked in the type of a term, but freshness, swapping, and name-abstraction play no role. Instead, Nanevski and Pfenning's system seems closer to the work of Hamana (in which the free names of a term are tracked in the type system) or of Odersky, described above, with explicit substitutions thrown in.

8.2.3 Reflection and Abstract Identifiers in Nuprl

Allen, Constable, Howe, and Aitken [6] introduced the idea of *reflected proof* in the Nuprl system, an interactive theorem proving system based on Martin-Löf's intuitionistic type theory [73]. They demonstrated its expressive power in proving properties of important algorithms such as matching. More recently, Barzilay, Allen, and Constable [12, 11, 13] have developed a new approach to reflection in Nuprl [23].

Their approach is to encode Nuprl terms as an explicit term data structure and use Nuprl's *quotient types* to equate terms up to an appropriate theory of equality, including α-equivalence and extensionality. In this approach, Nuprl type-checking judgments as *well-formedness judgments*, so this approach is similar to a specialized form of HOAS used in Coq [32], in which well-formedness judgments are required to exclude exotic terms. Nuprl's subset type constructor can be used to simplify reasoning about such validity constraints. It is possible that nominal abstract syntax could be used to simplify some aspects of this development of reflection in Nuprl.

Allen [5] has also studied the idea of *abstract identifiers* from the point of view of recordkeeping and maintaining formal content. The main problem considered is not name-binding, but maintaining the integrity of a possibly-changing linked set of formal documents with some well-formedness requirements (as is the case in a *formal digital library*, of which the theorem/proof library of Nuprl is one example.

8.2.4 Linc

Linc is the most recent in a line of a meta-logics developed by Miller, McDowell, Tiu, and Momigliano [74, 75, 81, 83, 88, 123] based on λ-term abstract syntax. There are important similarities between Linc and nominal logic.

The direct predecessors of Linc are $FO\lambda^{\Delta\mathbb{N}}$ and $FO\lambda^{\Delta\nabla}$. $FO\lambda^{\Delta\mathbb{N}}$, developed by McDowell and Miller [75], is a first-order meta-logic incorporating λ-terms, definitional clauses and induction over natural numbers. ($FO\lambda^{\Delta\mathbb{N}}$ is first-order in the sense that quantification over predicates is not allowed, only over types built up using data types and functions). McDowell and Miller it to encode a number of logics such as linear logic and to perform reasoning over weak higher-order abstract syntax encodings, including as an encoding of the operational semantics of an imperative language.

Miller [81] found that the $\forall$-quantifier was being used in two separate, incompatible ways in some potential applications of $FO\lambda^{\Delta\mathbb{N}}$: in particular, in order to reason about name-abstraction and scope-extrusion in the π-calculus, about fresh reference generation in imperative languages, or about fresh nonce generation in a security protocol, it seems desirable to use the quantifiers $\forall$ and $\exists$ to generate fresh parameters denoting (intuitively at least) distinct names. However, this is incompatible with a logical reading of these quantifiers, since $\forall x.\forall y.p(x,y) \supset \forall x.p(x,x)$ and $\exists x.p(x,x) \supset \exists x.\exists y.p(x,y)$. Thus, neither $\forall$ nor $\exists$ can be used to introduce *semantically distinct* fresh parameters in all situations, since there is no guarantee that two distinct parameters x, y will be instantiated with distinct values.

To remedy this, Miller and Tiu developed $FO\lambda^{\Delta\nabla}$ [83], a first-order meta-logic with lambda terms, definitions, and a *new-quantifier* ∇. They gave a sequent calculus for $FO\lambda^{\Delta\nabla}$ in which sequents $\Sigma : \Gamma \longrightarrow \mathcal{A}$ consist of a global parameter context Σ, a hypothesis context Γ, and a conclusion $\mathcal{A}$; the hypotheses and conclusion are pairs $\sigma \triangleright A$ of *local contexts* and formulas. The global context includes parameters introduced by $\forall$ on the right and $\exists$ on the left; the local contexts consist of parameters introduced by ∇ on *either the left or right*. That is, the rules for ∇ are

$$\frac{\Sigma : \Gamma, (\sigma, x) \triangleright B(x) \longrightarrow \mathcal{A} \quad (x \notin \sigma, \Sigma)}{\Sigma : \Gamma, \sigma \triangleright \nabla x.B(x) \longrightarrow \mathcal{A}} \nabla L$$

$$\frac{\Sigma : \Gamma \longrightarrow (\sigma, x) \triangleright B(x) \quad (x \notin \sigma, \Sigma)}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla x.B(x)} \nabla R$$

As a result, the ∇-quantifier is self-dual, just like the И-quantifier; moreover, it commutes with *all* the connectives $\forall, \exists, \wedge, \vee, \supset, \neg$, whereas И only commutes with propositional connectives. The ∇ quantifier can be used to describe the behavior of π-calculus transitions involving scope extrusion accurately, and to define the simulation and bisimulation relations on π-calculus terms, as well as to describe the behavior of an interpreter for a small programming language with $\forall$-quantified goals.

Momigliano and Tiu [88] extended $FO\lambda^{\Delta\mathbb{N}}$ to include reasoning by induction and co-induction. The resulting logic is called Linc and seems extremely powerful. Tiu [123] reports a version of Linc that includes the ∇ quantifier. Linc has been implemented in the Basic Linc (BLinc) system, and several examples including the π-calculus have been verified using its facilities for induction. However, BLinc is a proof tool, not a programming language, so a direct comparison with αProlog is not particularly illuminating. In particular, as a proof tool, the burden of choosing proof steps and substitution instances for quantifiers is on the user, not on the implementation.

The ∇-quantifier of Linc and the И-quantifier of nominal logic seem very closely related. Until recently, the absence of a sequent-style proof-theory for nominal logic or a model theory for Linc made it difficult to refine this intuition. Moreover, the absence of a denotational explanation for the ∇-quantifier makes it difficult to know just what the objects are that Linc reasons about, to an even greater extent than in ordinary higher-order or λ-term abstract syntax. As a concrete example of this difficulty, it is not clear whether Linc's ∇-bound names should be subject to weakening and exchange or not. Formulas such as $\nabla x, y.B\ x\ y \supset B\ y\ x$ or $B \supset \nabla x.B$, where x is not free in B, require explicit local context weakening and exchange principles for their proof. In the absence of a semantics explaining what ∇-bound names are, it is not obvious whether such rules are justifiable.

Recent developments have made a partial comparison possible. Gabbay and Cheney [41] introduced a sequent proof system for nominal logic, and subsequently I have developed an improved sequent proof system in Chapter 4 of this dissertation. These systems explicate the difference between nominal logic and Linc's ∇ in syntactic terms: in nominal logic, И-bound names have global scope within a judgment but may be excluded from terms using an explicit freshness predicate, whereas ∇-bound names in Linc are limited to the scope of the ∇. Gabbay and Cheney also addressed the relationship between ∇ and И by showing how to interpret $FO\lambda^{\nabla}$ (that is, the fragment of Linc without definitions or (co)induction) within nominal logic soundly and nontrivially. They gave a translation mapping $FO\lambda^{\nabla}$ sequents and formulas to nominal logic sequents and formulas preserving derivability. However, some formulas not provable in $FO\lambda^{\nabla}$, such as $B \supset \nabla x.B$, are translated to provable formulas, in this case $B' \supset \text{И}x.B'$, where B' is the translation of B. Similarly, the exchange property holds for И, but not necessarily for ∇. However, even though it is incomplete, the translation does suggest that it may be possible to find a semantics for the ∇-quantifier in terms of nominal sets.

Conversely, Linc's treatment of definitions, induction, and co-induction seems extremely powerful and general. It may be possible to incorporate these features into nominal logic. The resulting logic could be a powerful tool for reasoning about logics and programming languages via nominal abstract syntax: a *nominal logical framework*.

Chapter 9

Concluding Remarks

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.

—Bertrand Russell

My thesis is that *nominal logic programming is a powerful technique for programming with names and binding.* This dissertation offers evidence in favor of this thesis in many ways. In this concluding chapter, I will review the contributions presented herein and show how they justify my thesis in Section 9.1. I will outline what I see as the most important next steps in Section 9.2. And I will conclude in Section 9.3.

9.1 Contributions

This dissertation presents the results of my investigations of nominal logic Programming, including the design and implementation of a particular nominal logic programming language, αProlog, and the logical foundations, semantics and constraint solving problems for general nominal logic programming.

In Chapter 2, I described the αProlog programming language, a polymorphically typed logic programming language with built-in support for names, binding, and programming with nominal abstract syntax. I also presented a variety of interesting programs written in αProlog, including programs implementing substitution, typing, normalization, α-inequivalence, and closure conversion for the λ-calculus, as well as an operational semantics for the π-calculus. These programs are direct translations of informal paper specifications used by researchers and programmers, so their relative correctness is self-evident.

In Chapter 3, I developed a new theory of nominal sets, more general than the *finite-supported* nominal sets considered by Pitts and Gabbay, based on a generalized approach to classifying support sets as *small* and name-sets as *large*, using mathematical structures called *support ideals.* In addition, I developed a theory of syntax with bound names using nominal sets that is used throughout the rest of the dissertation.

In Chapter 4, I presented a new form of *nominal logic*, which includes a novel quantifier И, special symbols for names, and an axiomatization of the concepts of freshness, swapping, and name-abstraction or binding. I presented a semantics for nominal logic in terms of nominal sets as well as a novel sequent calculus for nominal logic, which uses a mixed context to represent some information about freshness. I also showed that the sequent calculus has the important property of cut-elimination.

In Chapter 5, I showed that the semantics and sequent calculus for nominal logic agree: that is, the soundness and completeness properties hold for nominal logic relative to its nominal set semantics. This result is the first of its kind. Pitts gave an an explicit counterexample to completeness for nominal logic relative to finitely-supported nominal sets, and I gave another counterexample to compactness. The use of support ideals in the development of nominal sets was the key to proving completeness. In addition, I proved a generalized form of Herbrand's Theorem for nominal logic in terms of Herbrand universes consisting of nominal terms built using the syntactic methods of Chapter 3.

In Chapter 6, I presented formal operational semantics (a state-transition system) and denotational semantics (a least Herbrand model construction) for Horn clause nominal logic programming. I showed that the operational semantics is sound and complete relative to the denotational semantics. The techniques employed are drawn from the semantics of constraint logic programming, but there are significant differences between nominal logic programming and CLP, so that existing theorems on the semantics of CLP cannot simply be reused. Therefore the results, if not all of the methods, of this chapter are new.

In Chapter 7, I studied nominal constraint solving. I showed that solving even a single freshness constraint, equation, or equivariance constraint is **NP**-hard, and the first two problems are in **NP**. I also developed algorithms for solving these constraint problems. These algorithms were shown to be correct: they find all possible correct answers and only correct answers. Finally, I discussed the shortcomings of these algorithms, efficient special cases, and possible directions for future work in this area.

9.2 Future Work

There are a number of interesting directions for future work.

- *Support for other forms of name-binding:* Some example programs we have written in Chapter 2 require checking that certain occurrences of names are *globally unique*. This phenomenon also arises in programming languages like C or assembly language, where some names have global scope and must be declared or defined at most once. It may be worthwhile to formalize *globally scoped names* within nominal logic. Another common use of names is *prefix-qualification*, as is often used in module or namespace systems. It may be of interest to find ways to better support prefix-qualified names and name-resolution within nominal logic. Also, recent versions of FreshML have provided the ability to abstract over general data structures such as lists, rather than single names. This behavior is not supported yet in nominal logic or αProlog.

- *Support for generic capture-avoiding substitution:* One really desirable feature provided by higher-order abstract syntax and Qu-Prolog is built-in capture-avoiding substitution. There is no problem in principle with introducing an explicit capture-avoiding substitution function symbol to αProlog, similar to that provided by Qu-Prolog, and I have experimented with this feature in the implementation of αProlog and found it to be very useful. This inspired the definition of generic capture-avoiding substitution in Section 3.5.5. However, the experimental implementation only performs substitution for closed terms, and unification does not take substitution into account (unlike in Qu-Prolog).

- *Complexity and efficiency issues:* According to [127], nominal pattern unification can be implemented in polynomial (quadratic) time. Efficient algorithms for nominal pattern unification need to be developed. In addition, efficient techniques for nominal constraint solving and practical techniques for dealing with name-name and equivariance constraints are desirable so that more general nominal logic programs can be executed efficiently.

- *Nominal equational unification:* In the π-calculus, often terms are considered up to a *structural equivalence* defined by ordinary associativity and commutativity laws as well as laws such as

$$\begin{array}{rcl} \nu a.P & = & P \quad (a \notin FV(P)) \\ \nu a.\nu b.P & = & \nu b.\nu a.P \\ (\nu a.P)|Q & \equiv & \nu a.(P|Q) \quad (a \notin FV(Q)) \end{array}$$

  Structurally equivalent process terms always have the same behavior, so much effort can be saved if computations can be specified in terms of structural equivalence classes rather than process terms. This is an interesting example of a *nominal equational unification* problem, that is, a problem of unifying terms modulo an equality theory specified in nominal logic. Nominal equational unification deserves further study.

- *A linear ordering on names:* One of the big disadvantages to equivariance is that there is no equivariant linear ordering on names (cf. Remark 4.2.12). Linear orderings are useful for efficient data structures and algorithms such as binary trees and binary searching. Can efficient name-based data structures be implemented in the presence of this limitation, or if not, is there a logically justifiable way to fix this?

- *Compilation and program analysis:* Currently αProlog is implemented as an interpreted language. What new compilation techniques, if any, are needed to compile αProlog programs? Can αProlog be implemented as an extension to an existing constraint programming framework such as CIAO Prolog [52]? Can known logic program analysis techniques, such as mode, determinism and termination checking, be extended to αProlog?

- *Semantic issues:* We presented a classical semantics for only the Horn clause fragment of αProlog. However, αProlog includes other features such as negation-as-failure which are not treated in this semantics. It would be desirable to extend the semantics to handle negation-as-failure. Also, the implementation of αProlog contains support for hereditary Harrop formulas. The semantics of hereditary Harrop formula logic programming is usually formulated in terms of uniform proofs, that is, using proof-theoretic semantics [82]. Therefore it would be interesting to develop such a semantics for αProlog, both to verify the correctness of the implementation and to make it easier to incorporate higher-order features into αProlog. The work of Leach, Nieva, and Rodríguez-Artalejo may be an appropriate starting point [68].

- *Nominal logical frameworks:* An important long-term goal for this work is to develop a logical framework for carrying out syntactic proofs of meta-theoretic properties of programming languages and logics. In Chapter 8, we compared nominal logic programming with other advanced approaches to programming with names and binding. Many of these techniques, in particular higher-order abstract syntax, are also used in automated reasoning, where the disadvantages of semantic and algorithmic complexity are magnified. Therefore, we believe that nominal abstract syntax could also be of use in automated reasoning. However, testing this claim will require developing the theoretical and practical foundations of a *nominal logical framework*, and attempting to solve problems which are difficult for existing techniques to solve. This is an important area for future work.

9.3 Conclusions

The concepts of names and name-binding are endemic in the study of logics and programming languages, and have challenged the brightest logicians and computer scientists. The idea of equality up to "safe" renaming of bound names is simple enough to be grasped intuitively by students and left implicit by researchers. However, computers cannot be taught in this intuitive way, but must be programmed to deal with names correctly using unambiguous rules. Gabbay and Pitts discovered a new theory of syntax with names and binding based on permutations, which I call nominal abstract syntax. Nominal abstract syntax is substantially simpler than prior approaches and therefore should be easier to automate.

In this dissertation, I have introduced a new logic programming paradigm, *nominal logic programming*, based on nominal abstract syntax. I presented a particular nominal logic programming language, αProlog, and provided several examples of programs that are easier to write in αProlog than in any other language. I provided a rigorous foundation for nominal logic programming based on a revised form of nominal logic, using existing semantic techniques from research on constraint logic programming and the study of Fraenkel-Mostowski independence results. I have also investigated the constraint-solving problems involved in nominal logic programming: in particular I presented algorithms for solving the problems and proved upper and lower complexity bounds (leaving one important upper bound open).

These are small first steps in the area of nominal logic programming, and much remains to be done. Nevertheless, this work justifies my claim that nominal logic programming is a powerful technique for programming with names and binding.

Bibliography

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

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[3] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[4] Michael H. Albert and John Lawrence. Unification in varieties of groups: nilpotent varieties. *Canadian Journal of Mathematics*, 46(6):1135–1149, 1994.

[5] Stuart F. Allen. Abstract identifiers, intertextual reference and a computational basis for recordkeeping. *First Monday*, 9(2), February 2004. `http://www.firstmonday.org/issues/issue9_2/allen/index.html`.

[6] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of Fifth IEEE Symposium on Logic in Computer Science*, pages 95–197. IEEE Press, 1990.

[7] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[8] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):842–862, 1982.

[9] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] H. P. Barendregt. *The Lambda Calculus*. North-Holland, 1984.

[11] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl, 2003. `http://www.cs.cornell.edu/eli/misc/hoas-paper.pdf`.

[12] Eli Barzilay and Stuart F. Allen. Reflecting higher-order abstract syntax in Nuprl. In *Proceedings of 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '02)*, pages 23–32, 2002.

[13] Eli Barzilay, Stuart F. Allen, and Robert L. Constable. Practical reflection in Nuprl. In P. Kolaitis, editor, *Proceedings of 18th IEEE Symposium on Logic in Computer Science*. IEEE, 2003. Short presentation.

[14] L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *Proceedings of the 13th International Conference on Concurrency Theory*, pages 209–225. Springer-Verlag, 2002.

[15] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE, IEEE Computer Society Press.

[16] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. *Journal of Functional Programming*, 2004. To appear.

[17] J. Cheney and C. Urban. System description: Alpha-Prolog, a fresh approach to logic programming modulo alpha-equivalence. In J. Levy, M. Kohlhase, J. Niehren, and M. Villaret, editors, *Proc. 17th Int. Workshop on Unification, UNIF'03*, pages 15–19, Valencia, Spain, June 2003. Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia. Technical Report DSIC-II/12/03.

[18] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, 2004. To appear.

[19] James Cheney. The complexity of equivariant unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, 2004. To appear.

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

[21] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, fifth edition, 2003.

[22] Jacques Cohen. Logic programming and constraint logic programming. *ACM Computing Surveys*, 28(1):257–259, 1996.

[23] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[24] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program*, 8(2):173–202, 1987.

[25] H. B. Curry. Grundlagen der kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536,789–834, 1930.

[26] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[27] John Darlington and Yike Guo. Constraint logic programming in the sequent calculus. In *Proceedings of the 1994 Conference on Logic Programming and Automated Reasoning (LPAR 1994)*, volume 822 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 1994.

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

[29] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[30] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34(5):381–392, 1972.

[31] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[32] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Int. Conf. on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag LNCS 902.

[33] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157(1–2):183–235, 2000.

[34] M. J. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin, Texas, 1979. Academic Press.

[35] Ulrich Felgner. *Models of ZF-Set Theory*. Number 223 in Lecture Notes in Mathematics. Springer-Verlag, 1970.

[36] Maribel Fernández, Murdoch Gabbay, and Ian Mackie. Nominal rewriting. In *Proceedings of the 6th Conference on Principles and Practice of Declarative Programming (PPDP 2004)*, 2004. To appear.

[37] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In Longo [70], pages 193–202.

[38] G. Frege. Begriffsschrift: a formula language, modeled upon that of arithmetic, for pure thought. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 1–82. Harvard University Press, 1967.

[39] M. J. Gabbay. *A Theory of Inductive Definitions with Alpha-Equivalence*. PhD thesis, University of Cambridge, 2001.

[40] M. J. Gabbay. Fresh logic: A logic of FM, 2003. Submitted.

[41] M. J. Gabbay and J. Cheney. A proof theory for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 139–148, Turku, Finland, 2004.

[42] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In Longo [70], pages 193–202.

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

[44] Murdoch Gabbay. A general mathematics of names in syntax. Submitted, March 2004.

[45] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[46] J. A. Goguen, J. W. Thatcher, E. Wagner, and J. B. Wright. Abstract data types as initial algebras and the correctness of data representations. In *Computer graphics, pattern recognition, and data structure*, pages 89–93. IEEE, 1975.

[47] Mikael Goldmann and Alexander Russell. The complexity of solving equations over finite groups. *Information and Computation*, 178:253–262, 2002.

[48] Makoto Hamana. 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, pages 243–262. Springer-Verlag, 2001.

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

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

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

[52] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-Garcá, and G. Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, 1999.

[53] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proceedings of the 1997 Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *LNCS*, pages 153–169, 1997.

[54] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In Longo [70], pages 204–213.

[55] Christopher J. Hogger. *Essentials of Logic Programming*, volume 1 of *Graduate Texts in Computer Science*. Clarendon Press, 1990.

[56] Furio Honsell and Marino Miculan. A natural deduction approach to dynamic logic. In S. Berardi et al, editor, *Types '95 Conf. Proc.*, volume 1158 of *Lecture Notes in Computer Science*, pages 165–182, 1996.

[57] John E. Hopcroft and Jeffrey D. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[58] Gérard Huet. The undecidability of unification in third-order logic. *Information and Control*, 22:257–267, 1973.

[59] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–67, 1975.

[60] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL 1987)*, pages 111–119. ACM Press, 1987.

[61] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[62] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46, 1998.

[63] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. Int. Conf. on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag New York, Inc., 1985.

[64] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 2004. Submitted.

[65] Michael Kohlhase, Susanna Kuschert, and Manfred Pinkal. A type-theoretic semantics for λ-DRT. In P. Dekker and M. Stokhof, editors, *Proceedings of the 10th Amsterdam Colloquium*, pages 479–498, Amsterdam, 1996. ILLC.

[66] J. Lambek and P. J. Scott. *Introduction to higher-order categorical logic*. Cambridge University Press, 1986.

[67] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory.* Springer-Verlag, 1992.

[68] Javier Leach, Susan Nieva, and Mario Rodríguez-Artalejo. Constraint logic programming with hereditary Harrop formulas. Preprint at `http://arxiv.org/abs/cs/0404053`, April 2004.

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

[70] Giuseppe Longo, editor. *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, 1999. IEEE, IEEE Press.

[71] Christopher Lynch and Barbara Morawska. Goal-directed E-unification. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, pages 231–245, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[72] A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[73] Per Martin-Löf. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory. Bibliopolis, 1984.

[74] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1997.

[75] Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic (TOCL)*, 3(1):80–136, 2002.

[76] Matías Menni. About И-quantifiers. *Applied Categorical Structures*, 11:421–445, 2003.

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

[78] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[79] Dale Miller. Unification of simply typed lamda-terms as logic programming. In Koichi Furukawa, editor, *Logic Programming, Proceedings of the Eighth International Conference*, pages 255–269, Paris, France, June 24–28 1991. MIT Press.

[80] Dale Miller. Abstract syntax for variable binders: an overview. In John Lloyd et al., editor, *Computational Logic - CL 2000*, number 1861 in LNAI. Springer, 2000.

[81] Dale Miller. Encoding generic judgments: Preliminary results. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, volume 58(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[82] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[83] Dale Miller and Alwen Tiu. A proof theory for generic judgments: extended abstract. In *Proc. 18th Symp. on Logic in Computer Science (LICS 2003)*, pages 118–127. IEEE Press, 2003.

[84] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus.* Cambridge University Press, 1999.

[85] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I-II. *Information and Computation*, 100(1):1–77, September 1992.

[86] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

[87] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 1996 Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[88] Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In *Proceedings of TYPES 2003*, volume 3085 of *LNCS*, pages 293–308. Springer-Verlag, 2003.

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

[90] G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Oxford University Press, 1998.

[91] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 206–217. ACM Press, 2002.

[92] Aleksandar Nanevski and Frank Pfenning. Meta-programming with names and necessity. Submitted, 2004.

[93] Sara Negri and Jan von Plato. *Structural Proof Theory.* Cambridge University Press, 2001.

[94] Anil Nerode and Richard A. Shore. *Logic for Applications.* Springer-Verlag, second edition, 1997.

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

[96] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer, 2002.

[97] Martin Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 48–59, January 1994.

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

[99] Christos H. Papadimitriou and Kenneth Stieglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover, 1998.

[100] Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of the 1992 International Conference on Logic Programming and Automated Reasoning (LPAR '92)*, number 624 in LNAI, pages 190–201, 1992.

[101] F. Pereira and D. H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Journal of Artificial Intelligence*, 13(3):231–278, 1980.

[102] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 17, pages 1063–1147. Elsevier Science, 2001.

[103] Frank Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[104] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89)*, pages 199–208. ACM Press, 1989.

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

[106] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[107] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

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

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

[110] Andrew Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science: Proc. 18th Int. Symp. MFCS '93*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.

[111] Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 391–405. Springer-Verlag, 1993.

[112] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. English translation appears in [113].

[113] M. Schönfinkel. On the building blocks of mathematical logic. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 355–366. Harvard University Press, 1967.

[114] Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In *Proceedings of the 2004 Computer Science Logic Conference*, 2004. To appear.

[115] C. Schürmann, R. Fontana, and Y. Liao. Delphin: Functional programming with deductive systems. Available at `http://cs-www.cs.yale.edu/homes/carsten/`, 2002.

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

[117] Mark R. Shinwell. Swapping the atom: Programming with binders in Fresh O'Caml. In Furio Honsell, Marino Miculan, and Alberto Momigliano, editors, *Proceedings of the Second ACM SIGPLAN Workshop on Mecahnized*

Reasoning about Languages with Binding (MERLIN 2003), pages 171–175, 2003.

[118] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[119] Wayne Snyder. *A Proof Theory for General Unification*, volume 11 of *Progress in Computer Science and Applied Logic*. Birkhäuser, 1991.

[120] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1–3):17–64, October-December 1996.

[121] J. Staples, P. J. Robinson, R. A. Paterson, R. A. Hagen, A. J. Craddock, and P. C. Wallis. Qu-Prolog: An extended Prolog for meta level programming. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 23. MIT Press, 1996.

[122] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[123] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications.* PhD thesis, University of Pennsylvania, Philadelphia, PA, May 2004.

[124] J. K. Truss. Permutations and the axiom of choice. In Richard Kaye and Dugald Macpherson, editors, *Automorphisms of First-Order Structures*, pages 131–152. Oxford, 1994.

[125] C. Urban and J. Cheney. Avoiding equivariance in alpha-Prolog. Submitted, 2004.

[126] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC)*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527, Vienna, Austria, 2003. Springer-Verlag.

[127] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 2004. To appear.

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

[129] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, Philadelphia, PA, August 1996.

[130] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *Proceedings of TYPES 2003*, pages 355–377, 2003.

[131] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. CLF: A dependent logical framework for concurrent computations. Submitted, April 2004.

Index