December 2006

A Review of Three Techniques for Formally Representing Variable Binding

Jeffrey A. Vaughan
University of Pennsylvania, vaughan2@seas.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-06-19.

A Review of Three Techniques for Formally Representing Variable Binding

Abstract
This paper compares three models for formal reasoning about programming languages with binding. Higher order abstract syntax (HOAS) uses meta-level binding to represent object-level binding [PE88]. Nominal Logic couples a concrete representation of bound variables with a formal apparatus for safely manipulating bound variables [Pit03]. The locally named binding representation places bound and free variables in different syntactic sorts [MP99]. This paper surveys each binding model, and compares it to the others and to Gordon and Melham's axiomatization of the untyped lambda calculus [GM97]. Comparisons are made based on expressive power, transparency to human readers, and suitability for mechanized reasoning of each binding model. Each system excels in one area; HOAS is most expressive, Nominal Logic most transparent, and locally named most mechanizable.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-06-19.

A Review of Three Techniques for Formally Representing Variable Binding

Jeffrey A. Vaughan

Technical Report Number MS-CIS-06-19
Department of Computer and Information Science
University of Pennsylvania

December 19, 2006

It's blatantly clear
You stupid machine, that what
I tell you is true
— Michael Norrish[1]

Abstract

This paper compares three models for formal reasoning about programming languages with binding. Higher order abstract syntax (HOAS) uses meta-level binding to represent object-level binding [PE88]. Nominal Logic couples a concrete representation of bound variables with a formal apparatus for safely manipulating bound variables [Pit03]. The locally named binding representation places bound and free variables in different syntactic sorts [MP99]. This paper surveys each binding model, and compares it to the others and to Gordon and Melham's axiomatization of the untyped lambda calculus [GM97]. Comparisons are made based on expressive power, transparency to human readers, and suitability for mechanized reasoning of each binding model. Each system excels in one area; HOAS is most expressive, Nominal Logic most transparent, and locally named most mechanizable.

1 Introduction

Many theoretical results in programming languages are published with very long, highly technical proofs. This material must be included for results to be believed. Unfortunately, the bookkeeping involved in developing such proofs can, in and of itself, burden researchers. Worse, a large volume of tedious detail can obscure the novel kernel of a paper. One solution to this problem may be writing machine checked proofs with tools such as Isabelle, Coq, or Twelf.

Curious whether automated theorem proving was ready for adoption in the programming languages community, Aydemir and colleagues [ABF$^+$05] posed the POPLmark challenge. In this

[1] From the website, Theorem Proving Haiku, at `http://www.cl.cam.ac.uk/Research/HVG/haiku.html`.

ongoing friendly contest, researchers mechanize proofs of a set of theorems using a variety of theorem provers. Some of these researchers gathered at the 2006 Progress on POPLmark meeting to share their experiences with the challenge. They found that representing variable binding is the hardest part of mechanizing programming languages metatheory[2].

This paper compares three techniques for representing variable binding in formal metatheory. I compare locally named, Nominal Logic, and higher order abstract syntax (HOAS) approaches to each other and to the well known binding models of de Bruijn and Barendregt. Throughout, I use the term *binding model* broadly to denote a style of encoding and reasoning about variable binding. Binding models are distinct from any particular theorem prover or logic, and are realizable in a variety of forms. Nominal logic, HOAS, and locally named are specific binding models.

These binding models were developed at different times by different groups and published with divergent presentation styles. So that we may see interesting distinctions clearly, I unify some technical details. For example, I renamed syntactic sorts, changed lists into sets, and slightly altered definitions. These changes do not affect the content of this discussion.

Unfortunately, the format of this paper constrains the material it can cover, thus many interesting topics are out of scope. While my interest in binding models is driven by mechanization, this paper does not discuss individual proof assistants, except to illustrate general points about the mechanizability of binding models or to explain particular technical quirks. Additionally, while I compare—and even grade—the binding models, different binding models may be appropriate in different circumstances; it does not make sense to declare one model best, and this paper does not do so. Lastly, there is a complex and varied ecosystem of binding models. Unfortunately, this paper ignores the weak higher order abstract syntax [DFH95] and locally nameless [MM04] models, as well as work relevant to implementers, such as Fresh Objective Caml [SP05, Shi05] and Cαml [Pot06]. Denotational semantics style investigations of binding [MO95] and Altenkirch's model, which quotients alpha-equivalence judgments with bijections on variable names [Alt02], are also neglected.

The rest of this paper is outlined below. Section 2 presents a technical prelude to the critical review. It begins with a discussion of formal logic, followed by an informal treatment of substitution (and associated issues) in lambda calculus. This illustrates problems encountered with binding. The section ends with discussions of the Barendregt variable convention and de Bruijn's nameless variable representation. Section 3 presents the Gordon-Melham axioms; these specify how binding should work, but not how to define it. Sections 4 through 6 discuss the HOAS, locally named, and Nominal Logic binding models. I conclude with a side-by-side comparison of the binding models.

2 Preliminaries

2.1 Metalogics and Object Languages

We will use formal logic to reason about programming languages. I will refer to programming languages under discussion as *object languages* and the surrounding logics as *metalogics*. To reduce confusion, metalogical statements are typeset with standard mathematical notation, and object language syntax is in a `fixed width` font. Using the notation of Section 3, for example, $\lambda x.\ x$ is a meta-level abstraction, `fn x. x` is concrete object syntax, and *Lam* "x" (*Var* "x") is abstract syntax.

[2]An informal transcript and slides are available from `http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/`.

Everywhere, except for Section 6, we will use higher order logic [Lei94] as our metalogic. Recall that in higher order logic, one may quantify over predicates ($\forall P\, x\,.\ P(x) \implies P(x)$, for example) and that functions may have higher order sorts like $(a \to b) \to c$. While logic provides a framework for reasoning, we want to go farther and actually represent object languages. To do so, we write a *signature* which specifies the names and sorts of objects we will reason about. Sorts are analogous to types. For example, the signature for a theory of real numbers might contain the sort *real* and the symbols

$$\begin{array}{rcl} 0 & : & \mathit{real} \\ 1 & : & \mathit{real} \\ + & : & \mathit{real} \to \mathit{real} \to \mathit{real}. \end{array}$$

2.2 Lambda Calculus and Substitution

Binding appears in many mathematical contexts. In this paper, we only discuss lambda calculus and closely related systems. This restriction is reasonable in light of Curry and Feys's observation that "any binding operation can in principle be defined in terms of functional abstraction"[3] [CFC58, 85]. Of course, lambda calculus is also interesting in its own right. In this section we will define the syntax of the untyped lambda calculus, substitution, alpha equivalence and beta reduction. The discussion of substitution illustrates the difficulty of working with binding.

We define the untyped lambda calculus using the following grammar:

$$\begin{array}{lrcl} \text{Variables} & x, y & ::= & \mathtt{x} \mid \mathtt{y} \mid \ldots \\ \text{Terms} & s, t, u & ::= & x \mid s\ t \mid \mathtt{fn}\ x.\ s \end{array}$$

Except for the abstraction notation, this definition is standard. We use $\mathtt{fn\ x.\ x}$ for object-level terms to distinguish them from meta-level terms like $\lambda x.\ x$. This distinction is important in Section 4 where we encounter lambda terms in both the metalogic and object language.

Three essential concepts in lambda calculus are substitution, alpha equivalence, and beta reduction. Substitution replaces all instances of some free variable with a term. We write $[x := s]t$ to mean t with s substituted for each free occurrence of x. Alpha equivalence relates terms which are identical up to the renaming of bound variables. For example,

$$\mathtt{fn\ x.\ z\ x} =_\alpha \mathtt{fn\ y.\ z\ y} \neq_\alpha \mathtt{fn\ x.\ w\ x}.$$

Intuitively, we would like alpha equivalent terms to be extensionally equal, that is, doing the same thing to two alpha equivalent terms should yield alpha equivalent results. Beta reduction computes the application of an abstraction to an argument. The formal definitions of both alpha equivalence and beta reduction depend on substitution.

Variable confusion makes substitution hard to define [Bar84]. To illustrate, let us compare several potential definitions of substitutions [Pie02]. The first and simplest is

$$\begin{array}{rcll} [x := s]x & \triangleq_\otimes & s & \\ [x := s]y & \triangleq_\otimes & y & \text{where } x \neq y \\ [x := s](t\ t') & \triangleq_\otimes & ([x := s]t)\ ([x := s]t') & \\ [x := s](\mathtt{fn}\ y.\ t) & \triangleq_\otimes & \mathtt{fn}\ y.\ [x := s]t. & \end{array}$$

[3] In some contexts the encoding of binding as functional abstraction is strained. ML type schemes, where sets of variables are bound simultaneously [PR05], provide one example of this. Cheney [Che05] examines several others and proposes a solution based on Nominal Logic. Such complications fall outside the scope of this paper.

(The symbol $\triangleq_{\otimes}$ indicates that the definition is incorrect and intended to illustrate a problem.) The definition is defective because, while $\texttt{fn x. x} =_{\alpha} \texttt{fn z. z}$ and

$$[\texttt{x} := \texttt{y}]\texttt{fn x. x} = \texttt{fn x. y},$$

we can also derive

$$[\texttt{x} := \texttt{y}]\texttt{fn z. z} = \texttt{fn z. z}.$$

Performing identical operations on alpha equivalent terms should yield alpha equivalent terms, but the results are clearly different! In this case, and in general, substituting for a bound variable is wrong [Bar84]. McKinna and Pollack call this *shadowing* [MP99].

A better substitution ignores abstractions when the abstraction and substitution variables are the same. That is:

$$[x := s](\texttt{fn } y.\ t) \quad \triangleq_{\otimes} \quad \begin{cases} \texttt{fn } y.\ t & \text{if } x = y \\ \texttt{fn } y.\ [x := s]t & \text{if } x \neq y \end{cases}$$

Unfortunately, this definition suffers from *capture*, where a free y in s may be accidentally bound [MP99, Win93]. Consider alpha equivalent terms $\texttt{fn y. x}$ and $\texttt{fn z. x}$. Applying substitution $[\texttt{x} := \texttt{y}]$ to each yields $\texttt{fn y. y}$ and $\texttt{fn z. y}$—different terms again.

Before moving to capture-avoiding substitution, we must precisely define the free variables of a term. In the following definition, we define a free variable function by structural recursion over the syntax of terms:

$$\begin{aligned} \mathit{fv}(x) &\triangleq \{x\} \\ \mathit{fv}(t\ s) &\triangleq \mathit{fv}(t) \cup \mathit{fv}(s) \\ \mathit{fv}(\texttt{fn } x.\ t) &\triangleq \mathit{fv}(t) \setminus \{x\} \end{aligned}$$

Only the abstraction case is interesting; observe that it ensures bound variables do not count as free. Using this definition of free variables, we again modify substitution.

$$[x := s](\texttt{fn } y.\ t) \quad \triangleq_{\otimes} \quad \begin{cases} \texttt{fn } y.\ t & \text{if } x = y \\ \texttt{fn } y.\ [x := s]t & \text{if } x \neq y \wedge y \notin \mathit{fv}(s) \end{cases}$$

This version of substitution, which we call *Barendregt's partial substitution*, does not violate our central expectation about alpha equality: When defined, the results of applying identical substitutions to alpha equivalent terms are alpha equivalent. The problem with this definition rests with definedness. Barendregt's partial substitution is undefined for $[\texttt{x} := \texttt{y}](\texttt{fn y. x})$. This is better, of course, than the previous definition of substitution, which handled the example incorrectly. Since Barendregt's substitution is a partial function, it requires that we work with alpha equivalence classes and implicit renamings—not syntactic terms. Thus, we must reject this definition, as we will define alpha equivalence in terms of substitution and must avoid such definitional cycles [Bar84].

Curry and Feys [CFC58, Bar84] define substitution by explicitly renaming variables when necessary to avoid confusion. Let $v_0, v_1, v_2 \ldots$ be the sequence of variables *fresh* from (not free in) s

and t in the following definition[4]:

$$\begin{array}{rcll} [x := s]x & \triangleq & s & \\ [x := s]y & \triangleq & y & \text{if } x \neq y \\ [x := s](t\ t') & \triangleq & ([x := s]t)\ ([x := s]t') & \\ [x := s](\texttt{fn}\ x.\ t) & \triangleq & \texttt{fn}\ x.\ t & \\ [x := s](\texttt{fn}\ y.\ t) & \triangleq & \texttt{fn}\ z.\ [x := s]([y := z]t) & \text{if } x \neq y \end{array}$$

where

$$z \quad = \quad \begin{cases} y & \text{if } y \notin fv(s) \cup fv(t) \\ v_0 & \text{otherwise} \end{cases}$$

While correct and mathematically precise, this definition of substitution has metatheoretical warts. In particular, the renaming operation is only well defined because we can pick a unique z. This requires that variables be ordered [CFC58] or that we have the axiom of choice[5].

With this satisfactory definition of substitution, we can now define alpha equivalence and beta reduction. Alpha equivalence is defined by the axiom scheme

$$\texttt{fn}\ x.\ t =_\alpha \texttt{fn}\ y.\ [x := y]t \quad \text{where } y \notin fv(t).$$

The side condition prevents us from deriving erroneous statements such as

$$\texttt{fn x. z x y} =_\alpha \texttt{fn y. z y y}.$$

The definition of beta reduction is even easier,

$$(\texttt{fn}\ x.\ t)\ s \rightarrowtail_\beta [x := s]t.$$

Because substitution is capture avoiding, this does not require a side condition. As a check, note that we can derive

$$(\texttt{fn x. fn y. x})\ \texttt{y} \rightarrowtail_\beta \texttt{fn z. y}$$

as expected.

2.3 Evaluation Criteria

For a binding model to be useful, it must be both expressive and convenient. Here, expressiveness means the formal power of a system. Merely being able to write down binding is not sufficient—a model must be powerful enough to prove interesting theorems. I address this by examining binding models through the critical lens of the Gordon-Melham axioms.

The Gordon-Melham axioms are five propositions sufficient to formalize many interesting properties of the untyped lambda calculus [GM97]. For each binding model, I define a theory of the untyped lambda calculus (or in the case of HOAS a slightly larger system) and check if this theory

[4]Only abstraction has changed, but I list the complete system of equations because this is the final, correct, definition.

[5]Set theory can be formalized either with or without the axiom of choice. One important consequence of this axiom is that, given an infinite set of unordered elements, you can select an element from the set [Sup72]. The sequence of v_is allows us to obtain a fresh z, regardless of how sets are axiomatized.

is consistent with the Gordon-Melham axioms. Thus, consistency with the Gordon-Melham axioms will be the yardstick by which we measure expressiveness. Section 3 discusses Gordon and Melham's work in detail.

Formal expressiveness is not the complete story. Indeed, we wish to avoid "the Turing tar-pit in which everything is possible but nothing of interest is easy" [Per82, 54]. That is, we must ensure that it is not merely possible, but also practical, to write proofs using our binding models. An ideal model is not only expressive, but also transparent and mechanizable.

I will say a binding model is transparent when statements in the model: are easy for humans to read and write [dB72], follow standard mathematical convention, and are accessible to non-specialists [ABF$^+$05]. The first and second conditions help readers establish the adequacy of formal theories [HC05, Mit96]. The benefits of the third condition are self evident.

A binding model is mechanizable when statements in the model can be easily encoded and manipulated by computer. This is useful both for theorists working with theorem provers and for language implementers translating formal notions into compiler and type checker code.

2.4 Conventional Binding Models

Two canonical binding models are the Barendregt variable convention and de Bruijn's nameless representation. Unfortunately, these are both deficient for formalizing languages.

There are two interpretations of the Barendregt variable convention. According to Barendregt himself, within a particular mathematical context (such as a proof or definition), all bound variables should be distinct from the free variables [Bar84]. Others, such as Pitts, mention a related convention where bound variables must also be mutually distinct. In either case, we reason about alpha-equivalence classes of terms by looking only at certain well-selected members of those classes. We can trust proofs in this style for two reasons. First, the variable convention guides us in our choice of representative class members. Second, human researchers avoid introducing functions and predicates that distinguish alpha equivalent terms; this discipline enables us to treat terms in an alpha equivalence class as interchangeable entities. However, there is no direct way to formalize reasoning based on such tacit assumptions [Pit03]. The Nominal Logic and locally named binding models can be read as methods of formalizing the spirit, if not the letter, of Barendregt.

While following Barendregt's variable convention is a classic technique for writing pencil-and-paper proofs, representing terms using de Bruijn's nameless notation is the traditional approach to mechanized metatheory. In the de Bruijn model, bound variables are replaced by integer indices indicating where they are bound. For example, the term (`fn f. fn x. f x`) is represented by (*Lam* (*Lam* (*App* 1 0))). De Bruijn notation is complicated for human readers, and it does not correspond to our informal notations about languages; it is not transparent. However, proofs have been successfully carried out in this style [dB72, Alt93]. Is it really so obtuse that we shouldn't just use it? While this is a matter of taste, many researchers are dissatisfied with pure de Bruijn notation:

> *. . . nameless terms are good for machine manipulation and metamathematical reasoning; for the human user they are not very convenient . . .*
>
> — Barendregt [Bar84, 581]

> *. . . the notational clutter [of de Bruijn indices] becomes quite a heavy burden even for fairly small languages . . .*
>
> — Aydemir and colleagues [ABF$^+$05, 55]

. . . terms in de Bruijn syntax are unfit for human consumption . . .
— McBride and McKinna [MM04, 2]

3 The Gordon-Melham Axioms

Gordon and Melham use only five axioms to prove several critical theorems about lambda terms, including induction and recursion principles [GM97]. We are primarily concerned with this work because it provides a technical framework for the critical evaluation of binding models. Consider some binding model. If a natural theory of lambda calculus in this model entails the Gordon-Melham axioms, then this theory will entail at least as many theorems as the Gordon-Melham theory. Thus, we would be confident that our binding model is expressive. Conversely, if a model is too weak to admit the Gordon-Melham axioms (or close variants thereof), it will be too weak to express some subset of induction, recursion, and alpha equivalence. Such a model can only produce stillborn theories.

Gordon and Melham give their axioms in a multi-sorted higher order logic. They define three sorts: *const* is the sort of constants, *string* the sort of variables, and *term* the sort of terms. Terms are built from the constructors

$$\begin{aligned} \mathit{Con} &: \mathit{const} \to \mathit{term} \\ \mathit{Var} &: \mathit{string} \to \mathit{term} \\ \mathit{App} &: \mathit{term} \to \mathit{term} \to \mathit{term} \\ \mathit{Lam} &: \mathit{string} \to \mathit{term} \to \mathit{term} \end{aligned}$$

and the signature includes the following additional operators:

$$\begin{aligned} \mathit{fv} &: \mathit{term} \to \mathit{string\ set} \\ [\_ := \_]\_ &: \mathit{term} \to \mathit{term} \to \mathit{string} \to \mathit{term} \\ \mathit{Abs} &: (\mathit{string} \to \mathit{term}) \to \mathit{term} \end{aligned}$$

Applications of the term constructors encode object-level terms in the obvious manner. For example,

$$\mathit{Lam}\ \text{“x”}\ (\mathit{Lam}\ \text{“y”}\ (\mathit{App}\ (\mathit{Var}\ \text{“x”})\ (\mathit{Var}\ \text{“y”})))$$

represents `fn x. fn y. (x y)`. The functions *fv* and $[\_ := \_]\_$ will calculate free variables and substitutions. We will discuss *Abs* with axiom GM5.

This treatment diverges from Gordon and Melham's original paper in three respects. First, they parameterized *term* by *const*. While doing so provides practical benefits from a mechanization standpoint, such benefits are orthogonal to our discussion, and I elide this complication. Second, I curried $[\_ := \_]\_$. Third, Gordon and Melham worked in the Cambridge HOL system, which includes theories of natural numbers, sets, and strings [GM93, HOL05]. Given a finite set of strings, they can construct a string not in the set. To make this explicit we add the symbol

$$\mathit{fresh} : \mathit{string\ set} \to \mathit{string}$$

to Gordon and Melham's signature and assume the *freshness axiom*,

$$\vdash\ \forall S : \mathit{string\ set}\,.\ \mathit{finite}(S) \implies \mathit{fresh}(S) \notin S. \tag{GM0}$$

3.1 Axioms for Substitution

This section presents Gordon and Melham's first three axioms, which address free variables, substitution, and alpha equivalence. The first axiom asserts that function *fv* calculates the free variables of a term.

$$\begin{aligned} \vdash\ & \forall k\,.\ fv(Con\ k) = \{\ \}\ \wedge \\ & \forall x\,.\ fv(Var\ x) = \{x\}\ \wedge \\ & \forall t\,u\,.\ fv(App\ t\ u) = fv(t) \cup fv(u)\ \wedge \\ & \forall x\,t\,.\ fv(Lam\ x\ t) = fv(t) \setminus \{x\} \end{aligned} \tag{GM1}$$

Except for constant terms, which do not occur in the standard treatment of lambda calculus, this definition matches our informal definition from Section 2.2.

Following Barendregt, Gordon and Melham exploit the free variable function to formally state the *substitution axiom*:

$$\begin{aligned} \vdash\ & \forall k\,u\,x\,.\ [x := u](Con\ k) = Con\ k\ \wedge \\ & \forall u\,x\,.\ [x := u]\,Var\ x = u\ \wedge \\ & \forall u\,x\,y\,.\ (x \neq y)\implies [x := u]\,Var\ y = Var\ y\ \wedge \\ & \forall t\,u\,v\,x\,.\ ([x := v](App\ t\ u)) = App\ ([x := v]t)\ ([x := v]u)\ \wedge \\ & \forall x\,t\,u\,.\ [x := u](Lam\ x\ t) = Lam\ x\ t\ \wedge \\ & \forall x\,y\,t\,u\,.\ (x \neq y) \wedge (y \notin fv(u))\implies [x := u](Lam\ y\ t) = Lam\ y\ ([x := u]t) \end{aligned} \tag{GM2}$$

As expected, the clauses pertaining to applications and variables match our informal definition. Even better, these clauses fully specify the operational behavior of substitution over their respective constructors.

Substitution over lambdas is more subtle. The first lambda case says substitution does not shadow bound variables; the second says it does not capture them. However, this axiom gives only a partial specification of substitution. That is, while GM2 tells us

$$\begin{aligned} [\text{“x”} := Con\ k](Lam\ \text{“y”}\ (Var\ \text{“x”})) &= Lam\ \text{“y”}\ ([x := Con\ k]\,Var\ \text{“x”}) \\ &= Lam\ \text{“y”}\ (Con\ k), \end{aligned}$$

it provides no information about

$$[\text{“x”} := Var\ \text{“y”}](Lam\ \text{“y”}\ (Var\ \text{“x”})).$$

Reading the abstraction clauses carefully, we find that the substitution axiom is the specification of Barendregt's partial substitution function from Section 2.2. There, I argued that the partial function was not sufficient to define substitution; this is still true. However, we are doing something different here. Axiom GM2 does not define $[\_ := \_]\_$. Rather, the axiom imposes a constraint on the function, but leaves some behavior unspecified. Models (i.e. implementations) of the Gordon-Melham axioms must provide a total substitution function like Curry and Feys's.

Gordon and Melham fully constrain substitution by introducing the *alpha equivalence axiom*. The axiom states

$$\vdash\ \forall x\,y\,t\,.\ y \notin fv(Lam\ x\ t)\implies Lam\ x\ t = Lam\ y\ ([x := Var\ y]t). \tag{GM3}$$

This corresponds to Barendregt's definition of alpha equivalence. Using axioms GM2 and GM3 together, we can simplify the preceding stuck substitution as follows:

$$\begin{aligned}
[\text{“x”} := \mathit{Var}\ \text{“y”}](\mathit{Lam}\ \text{“y”}\ (\mathit{Var}\ \text{“x”})) &= [\text{“x”} := \mathit{Var}\ \text{“y”}](\mathit{Lam}\ \text{“z”}\ ([\text{“y”} := \mathit{Var}\ \text{“z”}]\,\mathit{Var}\ \text{“x”})) \\
&= [\text{“x”} := \mathit{Var}\ \text{“y”}](\mathit{Lam}\ \text{“z”}\ (\mathit{Var}\ \text{“x”})) \\
&= \mathit{Lam}\ \text{“z”}\ ([\text{“x”} := \mathit{Var}\ \text{“y”}]\,\mathit{Var}\ \text{“x”}) \\
&= \mathit{Lam}\ \text{“z”}\ (\mathit{Var}\ \text{“y”}),
\end{aligned}$$

Note that picking fresh variable name “z” is an application of axiom GM0. This example illustrates a fact we will not prove—axioms GM0 through GM3 fully specify the behavior of substitution.

3.2 Axioms for Iteration

Using the completed specification of substitution, we define the *unique iteration axiom*, which provides a basis for deriving induction and recursion principles over lambda terms. The axiom follows:

$$\begin{aligned}
\vdash\ & \forall \mathit{con} : \mathit{const} \to \beta\,. \\
& \forall \mathit{var} : \mathit{string} \to \beta\,. \\
& \forall \mathit{app} : \beta \to \beta \to \beta \\
& \forall \mathit{abs} : (\mathit{string} \to \beta) \to \beta \\
& \exists! f : \mathit{term} \to \beta\,. \\
& \quad \forall k\,.\ f(\mathit{Con}\ k) = \mathit{con}(\mathit{Con}\ k)\ \wedge \\
& \quad \forall x\,.\ f(\mathit{Var}\ x) = \mathit{var}(\mathit{Var}\ x)\ \wedge \\
& \quad \forall t\,u\,.\ f(\mathit{App}\ t\ u) = \mathit{app}(ft)(fu)\ \wedge \\
& \quad \forall x\,u\,.\ f(\mathit{Lam}\ x\ u) = \mathit{abs}(\lambda y.\ f([x := \mathit{Var}\ y]u))
\end{aligned} \tag{GM4}$$

This states that we can uniquely define an iterative function, f, over lambda terms, using functions to specify f's behavior at each constructor. Each function has output sort β, which is a metavariable we can replace with, for example, *nat* or *string*. Functions *con* and *var* determine the result of applying f to abstract syntax tree leaves, and *app* calculates a β given the results of applying f to each branch of an application[6]. The abstraction case breaks the pattern, defines f in terms of $\mathit{abs} : (\mathit{string} \to \beta) \to \beta$, and ensures

$$f(\mathit{Lam}\ x\ u) = \mathit{abs}(\lambda y.\ f([x := \mathit{Var}\ y]u)).$$

Thus, to calculate over the structure of an abstraction, *abs* supplies its argument with a (fresh) variable name. (No catastrophe ensues if *abs* supplies a variable that is not fresh; but such an *abs* is unlikely to define an interesting f.)

We could not have given *abs* sort $\mathit{string} \to \beta \to \beta$. This is so because the resulting iteration principle could define functions which expose the bound variable of an abstraction. For example, consider function $\mathit{bound} : \mathit{term} \to \mathit{string\ set}$, which enumerates the bound variables in a term. The

[6]That *app* only examines returned βs and not the original *term*s distinguishes iteration from recursion.

following derivation shows that admitting *bound* leads to inconsistency:

$$\begin{aligned} \mathit{Lam}\ \text{“x”}\ (\mathit{Var}\ \text{“x”}) &= \mathit{Lam}\ \text{“y”}\ (\mathit{Var}\ \text{“y”}) \\ \mathit{bound}(\mathit{Lam}\ \text{“x”}\ (\mathit{Var}\ \text{“x”})) &= \mathit{bound}(\mathit{Lam}\ \text{“y”}\ (\mathit{Var}\ \text{“y”})) \\ \{\text{“x”}\} &= \{\text{“y”}\} \end{aligned}$$

The form of *abs* illustrates that there is a connection between certain functions of type $\mathit{string} \to \mathit{term}$ and lambdas. The *abstraction axiom* makes this explicit.

$$\vdash\ \forall x\, t\,.\ \mathit{Abs}\ (\lambda y.\ [x := \mathit{Var}\ y]t) = \mathit{Lam}\ x\ t \tag{GM5}$$

Thus an object-level abstraction is isomorphic to the meta-level function which produces the abstraction body when supplied with a bound variable. This axiom is necessary to iteratively define some functions, including the identity on terms. (To define the $\mathit{term} \to \mathit{term}$ identity we use *Abs* for *abs* in axiom GM4.)

3.3 Derived Theorems

Using the Gordon-Melham axioms, it is possible to prove interesting results about lambda terms. Importantly, we can derive the following induction theorem:

$$\begin{aligned} \vdash\ & \forall P : \mathit{term} \to \mathit{bool}\,. \\ & \quad \forall k\,.\ P(\mathit{Con}\ k)\ \wedge \\ & \quad \forall x\,.\ P(\mathit{Var}\ x)\ \wedge \\ & \quad \forall t\, u\,.\ P(t) \wedge P(u)\ \Longrightarrow\ P(\mathit{App}\ t\ u)\ \wedge \\ & \quad \forall x\, u\,.\ (\forall y\,.\ P([x := \mathit{Var}\ y]u))\ \Longrightarrow\ P(\mathit{Lam}\ x\ u) \\ & \Longrightarrow\ \forall u\,.\ P(u) \end{aligned}$$

This is a strong induction principle, because the induction hypothesis for the *Lam* case is strong. To see this, consider that under normal structural induction we would need to prove $P(\mathit{Lam}\ x\ u)$ given $P(u)$. Instead, this induction principal gives us $\forall y\,.\ P([x := \mathit{Var}\ y]u)$—that is, P holds regardless of our choice of bound variable.

Additionally, Gordon and Melham defined simultaneous renamings, iterated substitution, and a measure of term length. They proved a primitive recursion principle and that terms contain only finitely many free variables. Other researchers have continued this work, including Norrish who proved the substitution lemma, the Church-Rosser property for β, η, and $\beta\eta$ reductions, and the standardization lemma [Nor03].

3.4 Evaluation

The Gordon-Melham axioms provide a formal basis for automated reasoning about the untyped lambda calculus. Both Norrish and Gordon and Melham, found these axioms sufficient to prove interesting metatheoretical results.

Gordon and Melham selected their axioms for usability. For example, their alpha equivalence axiom (GM3) is derivable from the weaker statement

$$\vdash\ \forall y\, t\,.\ y \notin \mathit{fv}(t)\ \Longrightarrow\ \forall x\,.\ \mathit{Lam}\ x\ t = \mathit{Lam}\ y\ ([x := \mathit{Var}\ y]t).$$

However, Gordon and Melham rejected this potential axiom because it is inconvenient. Similarly, the abstraction axiom (GM5) is definable using axioms GM0–GM4; Gordon and Melham state it as an axiom due to its utility. If Gordon and Melham had given the minimal axiomatization of their system, most theories would include a boilerplate prelude deriving axioms GM3 and GM5.

Unfortunately, this work is dependent on the Cambridge HOL framework. We saw that Gordon and Melham rely on properties of HOL's *string* theory. Additionally, HOL includes the axioms of choice and excluded middle—it is not clear to what extent developments using the Gordon-Melham axioms are dependent on these. This is important because not all formalisms or proof assistants include these axioms. Despite this metatheoretical blemish, their theory is well defined, and they prove their axioms are consistent using a locally nameless variant of de Bruijn as a binding model [MM04].

4 Higher Order Abstract Syntax

In the higher order abstract syntax (HOAS) binding model, object language binding constructs are encoded as meta-level lambda abstractions [PE88]. The abstract syntax $Lam\ (\lambda x.\ Lam\ (\lambda y.\ App\ x\ y))$ will, for example, represent the object term `fn x. fn y. (x y)`. In the abstract syntax, the xs and ys are bound metavariables; in the object term the `x`s and `y`s are concrete variable names. As we will see, a key benefit of HOAS is that it allows us to write rules without requiring syntactic side conditions to forestall variable capture.

4.1 Case Study

For the rest of this section, we will work with a simple functional language which augments lambda calculus with conditionals and let binding.

$$\begin{array}{rcl} s, t, u & ::= & x \mid t\ u \mid \texttt{fn}\ x.\ t \\ & & \mid \texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \\ & & \mid \texttt{let}\ x_1 = t_1 \texttt{and}\ x_2 = t_2\ \texttt{and}\ \ldots \texttt{and}\ x_n = t_n\ \texttt{in}\ u \end{array}$$

I intend for the `let` construct to bind variables $x_1 \ldots x_n$ in u, but not in any t_i. In contrast, `if`—the standard ternary operator—binds nothing.

Encoding the above in HOAS will illustrate a general principle of the model. Object programs (or program fragments) are encoded as meta-level lambda calculus terms and syntactic sorts correspond to meta-level types. Particular pieces of object-level syntax map to meta-level constants. Lexical terminals (e.g. variables) are first order constants. Operators (e.g. `if`) are second order constants. And, binding operators (e.g. `let`) are third order constants[7].

It is this encoding of binders that gives HOAS its name. Pfenning and Elliot define these constants in polymorphic lambda calculus with pairs and pattern matching. While logics, including Church's, do not always include these features [Chu40], Pfenning and Elliot found them essential for practical encodings.

[7]Here we follow the convention that first order constants are base values, second order constants correspond to functions over base types, and third order constants have types like $(term \to term) \to term$ [HL78].

Our encoding is over the signature

$$\begin{aligned} \mathit{Con} &: \mathit{const} \to \mathit{term} \\ \mathit{Lam} &: (\mathit{term} \to \mathit{term}) \to \mathit{term} \\ \mathit{App} &: \mathit{term} \to \mathit{term} \to \mathit{term} \\ \mathit{Let} &: (\alpha \to \mathit{term}) \to \alpha \to \mathit{term} \\ \mathit{If} &: \mathit{term} \to \mathit{term} \to \mathit{term}. \end{aligned}$$

Note that both *Lam* and *Let* are higher order functions; we use them to represent object-level binders as meta-level functions. The *Let* constructor is polymorphic and the α in its type is implicitly quantified (we drop the $\forall$ to avoid confusion with the logical operator). We only consider closed object terms and do not include a constructor for variables; all object variables translate into bound variables in the metalogic. The following function encodes objects in HOAS:

object syntax		HOAS term	
$[\![k]\!]$	$=$	$\mathit{Con}\ k$	(constants)
$[\![x]\!]$	$=$	x	(variables)
$[\![t\ u]\!]$	$=$	$\mathit{App}\ [\![t]\!]\ [\![u]\!]$	
$[\![\texttt{fn}\ x.\ t]\!]$	$=$	$\mathit{Lam}\ (\lambda x.\ [\![t]\!])$	
$[\![\texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u']\!]$	$=$	$\mathit{If}\ [\![t]\!]\ [\![u]\!]\ [\![u']\!]$	
$[\![\texttt{let}\ x_1 = t_1\ \texttt{and}\ \ldots\ \texttt{and}$ $x_n = t_n\ \texttt{in}\ u]\!]$	$=$	$\mathit{Let}\ (\lambda\langle x_1, \ldots, x_n\rangle.\ [\![u]\!])\ \langle [\![t_1]\!], \ldots, [\![t_n]\!]\rangle)$	

We need not define substitution; HOAS provides it for free. For example, we can build a reduction relation, which subsumes $\rightarrowtail_\beta$, out of metalogic substitution and application. Some (but not all) of the necessary rules are:

$$\begin{aligned} \mathit{App}\ (\mathit{Lam}\ t)\ u &\rightarrowtail t(u) \\ \mathit{Let}\ t\ u &\rightarrowtail t(u) \\ \mathit{If}\ \mathit{true}\ t\ u &\rightarrowtail t \\ \mathit{If}\ \mathit{false}\ t\ u &\rightarrowtail u \end{aligned}$$

For example, let's try reducing `let x = 4 and y = 2 in plus(x, y)`. Encoding this in HOAS and considering the reduction relation shows

$$\begin{aligned} \mathit{Let}\ (\lambda\langle x, y\rangle.\ \mathit{plus}\ x\ y)\ \langle 4, 2\rangle &\rightarrowtail ((\lambda\langle x, y\rangle.\ \mathit{plus}\ x\ y))\langle 4, 2\rangle \\ &=_{\beta\eta} \mathit{plus}\ 4\ 5 \end{aligned}$$

Here we gave `let` two arguments, but, because it is polymorphic, we could have given in any number. The $\beta\eta$ equality in the last step is equality over meta-level lambda terms which we exploit to get the correct object-level reduction. Definitions like these are the raison d'être of HOAS. They can be written without syntactic side conditions, a free variable function, or a definition of object-level substitution.

While substitution and reduction are easily defined, terms with free variables are more difficult. One way to encode `fn x. y` is as a meta-abstraction returning a term,

$$\lambda y.\ \mathit{Lam}\ (\lambda x.\ y).$$

This has the obvious deficiency that the encodings of open terms have a different syntactic sort than closed terms. Another approach is to insert metavariable $x : term$ to the metalogic's context. While this is effective, it requires complex meta-meta-logical reasoning to establish that the term encoding is adequate [HHP93, HL06].

4.2 Evaluation

The HOAS binding model is quite different from the normal pencil-and-paper conventions. While it introduces significant simplifications over traditional proofs—substitution becomes implicit, application trivial, and variable freshness irrelevant—HOAS also adds complications. Defining induction is challenging [ACM02], and Pfenning and Elliot do not mention inductive proofs at all in [PE88]. Additionally, proofs must be written over $\alpha\beta\eta$ equivalence classes of meta-level lambda terms [HC05]; this too adds complexity. While some practitioners find HOAS convenient, its significant departure from standard mathematical practice and consequent steep learning curve yield a non-transparent binding model.

HOAS has been successfully used in a variety of mechanized settings. Pfenning used HOAS and Twelf, the canonical theorem prover for HOAS metatheory, to formalize a variety of theorems about Mini-ML including compiler correctness, value soundness, and type preservation [PS99, Pfe]. Lee, Crary, and Harper formalized a semantics for Standard ML using HOAS in Twelf [LCH07].

Despite these great successes, mechanizing HOAS is not easy. Because meta-level abstractions (identified only up to β- or $\beta\eta$-equivalence) are a component of abstract syntax terms, structural induction on these terms is subtle. Twelf requires a special purpose layer above the normal metalogic for inductive reasoning. Limitations in the meta-meta-logic makes it impossible to write logical relations proofs in a natural fashion [HC05, ACM02]. Recently Ambler and colleagues implemented a HOAS system, called Hybrid, in Isabelle/HOL. In Hybrid, and in earlier work, they derived structural HOAS induction principles and even took steps toward automatically generating them [MAC01, ACM02]. While Ambler and colleagues appear to have solved some of the problems surrounding induction, they have not yet published a definitive account of their work. Additionally, mechanized HOAS approaches rely on higher order unification, which is undecidable. Fortunately, there are good incomplete algorithms for this [Hue75]. While HOAS can be successfully mechanized, such mechanization is complicated and the published results are not completely satisfactory.

HOAS is highly expressive. Pfenning and Elliot's presentation includes alpha equivalence, substitution, and an infinite supply of variables, for free. Hence HOAS can easily derive axioms GM0, GM2 and GM3. Additionally, it's quite easy to define inductive relations in their system. Both Hybrid and Twelf provide HOAS users with structural induction. Iterative functions are defined in Twelf using logic programming [PS02] and in Hybrid using Hilbert choice. This illustrates that HOAS is consistent with axiom GM4, but does not intrinsically entail it. While Pfenning and Elliot only represent closed terms, Ambler and colleagues use a distinguished sort of names as free variables [ACM02]; either appears consistent with GM1. Lastly, the key idea of HOAS is that object abstractions are represented as meta-abstractions. Axiom GM5 follows from the definition of *Lam*. Thus, informally, the HOAS theory of the lambda calculus is consistent with the Gordon-Melham axioms. In summary, while HOAS is very expressive, it is difficult to mechanize and difficult for non-experts to understand.

HOAS	
Expressiveness	**A-**
Mechanizability	**B**
Transparency	**C**

5 Locally Named

Next we consider the locally named binding model described by McKinna and Pollack [MP99]. In this binding model, free and bound variables live in distinct syntactic sorts. This makes substitution easy to formalize because well formed terms will automatically avoid variable shadowing and capture.

5.1 Case Study

McKinna and Pollack formalize pure type systems, but, for consistency, we restrict this discussion to the untyped lambda calculus. We will define our theory using four sorts. Sorts *term* and *const* are as usual, but sorts *var* and *param* are novel. Members of *var* represent bound variables and members of *param* represent free variables, or parameters. We write variables as v or w and parameters as p or q. All variable and parameter sorts contain an infinite number of elements.

We define terms over the following signature:

$$\begin{aligned} \mathit{Con} &: \mathit{const} \to \mathit{term} \\ \mathit{Var} &: \mathit{var} \to \mathit{term} \\ \mathit{Param} &: \mathit{param} \to \mathit{term} \\ \mathit{Lam} &: \mathit{var} \to \mathit{term} \to \mathit{term} \\ \mathit{App} &: \mathit{term} \to \mathit{term} \to \mathit{term} \end{aligned}$$

For example, we represent `fn v. p v` as $\mathit{Lam}\ v\ (\mathit{App}\ (\mathit{Param}\ p)\ (\mathit{Var}\ v))$. Because this is a concrete representation, we can easily define induction and over the structure of terms. However, we will find a pair of stronger induction principles shortly. Additionally, we augment the signature with symbols for calculating a term's free parameters and constants.

$$\begin{aligned} \mathit{params} &: \mathit{term} \to \mathit{param}\ \mathit{set} \\ \mathit{consts} &: \mathit{term} \to \mathit{const}\ \mathit{set} \end{aligned}$$

The *params* function is defined recursively by

$$\begin{aligned} \mathit{params}(\mathit{Param}\ p) &\triangleq \{p\} \\ \mathit{params}(\mathit{Var}\ v) &\triangleq \{\} \\ \mathit{params}(\mathit{Con}\ k) &\triangleq \{\} \\ \mathit{params}(\mathit{Lam}\ v\ t) &\triangleq \mathit{params}(t) \\ \mathit{params}(\mathit{App}\ t\ t') &\triangleq \mathit{params}(t) \cup \mathit{params}(t'). \end{aligned}$$

This is simpler than the informal definition of free variables in Section 2.2 because in the *Lam* case we can simply ignore bound variable v. The definition of *consts* is, mutatis mutandis, the

same. We need not add a function symbol $vars : term \to var$ as we plan to work only with closed terms—those terms for which *vars* would behave as the constant function yielding {}.

Next we define two flavors of substitution: one for parameters and one for variables. To do so we add symbols

$$\begin{array}{rcl} [\_ := \_]_p \_ & : & param \to term \to term \\ [\_ := \_]_v \_ & : & var \to term \to term \end{array}$$

to the signature. Defining the first is simple.

$$\begin{array}{rcl} [p := t]_p Param\ q & \triangleq & \begin{cases} t & p = q \\ Param\ q & \text{otherwise} \end{cases} \\ [p := t]_p Var\ v & \triangleq & Var\ v \\ [p := t]_p Con\ k & \triangleq & Con\ k \\ [p := t]_p App\ s\ u & \triangleq & App\ ([p := t]_p s)\ ([p := t]_p u) \\ [p := t]_p Lam\ v\ u & \triangleq & Lam\ v\ [p := t]_p u \end{array}$$

This is just textual substitution and equivalent to our first rejected definition from Section 2.2; we did not attempt to avoid variable capture or shadowing. Shadowing is not an issue, because parameters cannot be bound. However, as illustrated by the following example, variable capture still appears problematic:

$$[p := Var\ v]_p(Lam\ v\ (Param\ p)) = Lam\ v\ (Var\ v)$$

Except for the following abstraction case, the definition of $[\_ := \_]_v \_$ is similar.

$$[v := t]_v Lam\ w\ u \quad \triangleq \quad \begin{cases} Lam\ w\ u & v = w \\ Lam\ w\ [v := t]u & \text{otherwise} \end{cases}$$

While this substitution avoids shadowing, it too allows variable capture.

We want to avoid capturing uses of substitution, and it is obvious we can do so by only substituting terms containing no free *var*s. This observation must remain informal, as variable capture itself is an informal notion. From here on, we will only consider terms without free variables to be good terms.

To formalize variable closed terms, McKinna and Pollack define the *Vclosed* inductive relation:

$$\frac{}{Vclosed(Param\ p)}\ \textsc{Vcl-Param} \qquad \frac{}{Vclosed(Con\ k)}\ \textsc{Vcl-Const}$$

$$\frac{Vclosed([v := Param\ p]_v t)}{Vclosed(Lam\ v\ t)}\ \textsc{Vcl-Bind} \qquad \frac{Vclosed(t) \qquad Vclosed(u)}{Vclosed(App\ t\ u)}\ \textsc{Vcl-App}$$

Rule Vcl-Bind is key. This states that an abstraction with bound variable v is closed if its body is closed when v is replaced by a parameter. Because this relation is inductively defined, we can write proofs by induction over the derivation showing that a term is *Vclosed*. This is a weaker form of induction than Gordon and Melham's.

To recover the full strength of Gordon and Melham's induction principle, McKinna and Pollack introduce a new relation *aVclosed*. Except for the binding case, this relation is defined by the same rules as *Vclosed*.

$$\frac{\forall p\,.\ aVclosed([v := Param\ p]_v t)}{aVclosed(Lam\ v\ t)}\ \textsc{aVcl-Bind}$$

Induction over the structure of *aVclosed* is more powerful than over *Vclosed*. When inducting over the structure of an *aVclosed* derivation, the induction hypothesis says abstraction bodies are *aVclosed* closed regardless of how we replace the bound variable. This is equivalent to the derived Gordon-Melham induction principle. In contrast, the induction hypothesis for *Vclosed* derivations only says abstraction bodies are *Vclosed* when substituting with some particular parameter.

McKinna and Pollack prove that the two relations contain the same terms and correspond to the set of terms with no free variables. While *aVclosed* is better for induction, *Vclosed* is useful as an introduction form. Thus we can prove $Vclosed(t)$ using that relation's weak premises, and conclude that $P(t)$ holds, where P is a property proved inductively using *aVclosed*'s strong induction principle.

We are now ready to define beta reduction. It is simply

$$App\ (Lam\ v\ t)\ s \rightarrowtail [v := s]_v t \qquad \text{where } Vclosed(s).$$

The side condition ensures that the substitution will be capture avoiding. Section 2.2's informal β rule had no side conditions. But only *Vclosed* terms matter, and, for such terms, this rule matches the informal one.

McKinna and Pollack treat alpha equivalence as a new relation distinct from logical equality. The following inference rules identify alpha equivalent terms:

$$\frac{}{Var\ p =_\alpha Var\ p} \qquad \frac{}{Con\ k =_\alpha Con\ k} \qquad \frac{s =_\alpha s' \qquad t =_\alpha t'}{App\ s\ t =_\alpha App\ s'\ t'}$$

$$\frac{[v := Param\ p]_v t =_\alpha [v' := Param\ p]_v t'}{Lam\ v\ t =_\alpha Lam\ v'\ t'} \quad p \notin params(t), params(t')$$

Except for abstraction, each rule is simple. Two lambda terms are alpha equivalent when their bodies are alpha equivalent after substituting a fresh parameter for their bound variables. A nice result is that, in this setting, we can alpha rename any abstraction such that any variable is the outer most bound variable. That is,

$$Vclosed(Lam\ v\ t) \implies \forall w\,.\ \exists s\,.\ Lam\ w\ s = Lam\ v\ t$$

This provides more naming flexibility than our informal system where, for example, we can never rename `fn x. y` to make `y` the bound variable. This additional flexibility is a direct consequence of the locally named binding model.

Unfortunately there is a significant defect in the locally named treatment of alpha equivalence. The indicator function

$$f(t) = \begin{cases} 0 & \text{if } t = Lam\ v\ (Var\ v) \\ 1 & \text{otherwise} \end{cases}$$

is perfectly legal in the locally named model. This means that it is possible to distinguish two alpha equivalent terms. In turn, given $P(t)$ and $t =_\alpha t'$ we *cannot* conclude $P(t')$. If wish to

reason about alpha equivalence classes, we must do so explicitly. From a practical standpoint, this is substantial defect. While pencil-and-paper proofs using the Barendregt variable convention also suffer from this defect, normal practice allows us to gloss over it. Such are the benefits of informality. As discussed by McBride and McKinna, the closely related locally nameless representation, where alpha-equivalent terms are logically equivalent, provides a formal solution to this problem [MM04].

5.2 Evaluation

McKinna and Pollack's locally named representation is reasonably easy for humans. While using two sorts of variables is unconventional, it has appeared in earlier literature [Coq91]. Additionally, the scheme is similar to the Barendregt variable convention. In either case, free and bound names are drawn from distinct sets; McKinna and Pollack are simply more explicit about this. Their notation is easy to read and locally named lambda terms look similar to informal ones. Additionally, their work does not depend on esoteric mathematics or logic and should be readily accessible to non-specialists. One defect in the system is that alpha equivalent terms are not logically equal, and users must sometimes explicitly reason about alpha equivalence where they would not do so on paper. However, the locally named binding model is, overall, quite transparent.

The locally named representation is sufficient to prove most, but not all, the Gordon-Melham axioms. Locally named terms are built from first order constructors. Consequently, we can easily define iterative functions and derive axiom GM4. Induction over *aVclosed* is isomorphic to the Gordon-Melham induction principle. Substituting *Vclosed* terms fulfills Gordon and Melham's substitution axiom (GM2), and *params* satisfies the free variable axiom (GM1). Locally named is also consistent with GM0. Unfortunately, treating alpha equivalence as an auxiliary relation does not let us derive axiom GM3; worse, because alpha-equivalent terms can be distinguished, locally named is inconsistent with axiom GM3. Note this inconsistency corresponds to the above practical expressiveness problem.

The locally named approach is readily mechanizable. Language definitions do not rely on complicated logic features. Indeed, little more is required than a countably infinite set, the ability to define recursive datatypes, and theory of inductively defined relations.

Both humans and computerized proof assistants can easily understand the locally named representation. Unfortunately, alpha equivalence is second class, and this diminishes the expressive power of the binding model.

Locally Named	
Expressiveness	**B**
Mechanizability	**A**
Transparency	**A-**

6 Nominal Logic

Our next binding model, Pitts's Nominal Logic [Pit03], represents a significant departure from the previous models. Before, we defined object language theories in standard logics; here we will define a new logic to facilitate reasoning about object languages.

Nominal Logical is an extension of multi-sorted first order logic with equality. Recall that in first order logic, quantifiers may not range over functions or predicates [Dav93, GLM97]. Nominal Logic adds three essential features to first order logic: atoms, swapping and a freshness predicate.

In this model, variable names are called *atoms* and inhabit distinguished sorts. The logic contains functions to *swap* atoms and to test if an atom is *fresh* with respect to an object. While such operations can be defined in higher order logic [UT05] or the Calculus of Inductive Constructions [AB06], such work is outside the scope of this paper.

6.1 Distinguishing Features

The key idea of Nominal Logic is that substitution and renaming are the wrong primitives for manipulating variables. This is because renaming can affect many useful propositions. For example, given distinct atoms a and b, it is true that $a \neq b$ but renaming a to b gives $b \neq b$, which is clearly false. In contrast, swapping a and b yields $b \neq a$; this is still a true statement.

In Nominal Logic, we will only work with *equivariant* predicates: those whose validity is unaffected by swapping. All standard logical operators, including equality, negation, conjunction, disjunction, existential quantification, and universal quantification are equivariant. Additionally, the greatest and least fixed points of monotone operators are equivariant. Equivariant operators are compositional; a predicate built from equivariant operators is itself equivariant. This is useful because we will leverage equivariance to, as in informal proofs, reason about entire alpha equivalence classes using individuals from those classes as concrete representatives.

6.2 Some Rules of Nominal Logic

Nominal Logic sorts are grouped into two flavors, atom sorts and data sorts. Atom sorts contain atoms (i.e. variable names), and data sorts contain everything else. Theories may have multiple sorts of each variety. For example, a typed language might be represented with two atom sorts, one for term variables and one for type variables. By convention, the symbol A ranges over the set of atom sorts and S ranges over the set of all sorts. The variables a, b, and c range over atoms.

In addition to the standard logical symbols (like equality and disjunction), Nominal Logic has symbols for swapping and freshness. These are written as follows[8]:

$$\begin{aligned} (\_\ \_) \bullet \_ &: A, A, S \to S \\ \_ \sharp \_ &: \text{relation over } (A, S) \end{aligned}$$

We pronounce $(a\ b) \bullet x$ as "swap a for b in x" and $a \sharp x$ as "a is fresh for x." We will introduce additional symbols shortly, but they can be defined (with a some additional axioms) in terms of the above.

Although Pitts's logic is fully axiomatized, we will examine a mix of axioms and theorems; this is more illuminating than an exhaustive list of axioms.

We begin with swapping. The swap operator is characterized by the following equations:

$$(a\ b) \bullet c = \begin{cases} a & c = b \\ b & c = a \\ c & \text{otherwise} \end{cases}$$

$$(a\ b) \bullet f(x_1, \ldots, x_n) = f((a\ b) \bullet x_1, \ldots, (a\ b) \bullet x_n)$$

[8] Changing from curried functions to functions of multiple arguments is necessary to work within first order logic.

The first equation corresponds to the our intuitive notion of swapping atoms. The second is more interesting. While this certainly seem reasonable for constructors, we can imagine functions where it does not hold. For example, consider the problematic function

$$bad(x) = \begin{cases} 0 & x = a \\ 1 & \text{otherwise} \end{cases}.$$

Evaluating *bad* before swapping gives

$$(a\ b) \bullet bad(a) = (a\ b) \bullet 0 = 0,$$

but swapping first yields

$$(a\ b) \bullet bad(a) = bad((a\ b) \bullet a) = bad(b) = 1.$$

The problem here is that the *bad* is not equivariant; we cannot define such functions within Nominal Logic. While it is occasionally necessary to define functions by adding axioms, it's necessary to check any new axioms for consistency—*bad* fails that check and is inconsistent with Nominal Logic.

Pitts derives the following equivariance property: for any predicate P,

$$\vdash\ \forall a\, b\, x_1 \ldots x_n\,.\ P(x_1, \ldots, x_n) \iff P((a\ b) \bullet x_1, \ldots, (a\ b) \bullet x_n).$$

This definition allows us to swap variables throughout a proposition with out changing the proposition's validity.

We proceed to freshness. Distinct atoms are fresh with respect to each other.

$$\vdash\ \forall a\, b\,.\ a \sharp b \iff a \neq b$$

Additionally, the is-fresh-for relation can be lifted over functions, with the following property:

$$\vdash\ \forall a\, x_1 \ldots x_n\,.\quad a \sharp x_1 \wedge a \sharp x_2 \wedge \ldots \wedge a \sharp x_n \implies a \sharp f(x_1, \ldots, x_n)$$

This states that if an atom is fresh from a function's arguments, that atom is fresh from the function's result. However the converse is *not* true. Our final property is the *fresh variable axiom*, which declares that there a variable fresh from any list of terms. The axiom[9] is

$$\vdash\ \forall x_1 \ldots x_n\,.\ \exists a\,.\ a \sharp x_1 \wedge \ldots \wedge a \sharp x_n.$$

While the above definitions suffice to formalize lambda calculus, and likely most other languages with binding, Pitts introduces a binding construct directly to the logic. The purpose of this is to, HOAS like, define binding once and for all, thus eliminating boilerplate definitions from object theories. Following Pitts, we add a new meta-operation on sorts, and let $[A]S$ represent the sort of A abstractions over S. Objects of this sort are written $a.\,s$ where $a : A$ and $s : S$. (Pitts makes $\_.\,\_$ a new syntactic form, but a family of function symbols would work too.) For example, we will represent (`fn a. b a`) by $Lam\ (a.\,App\ (Var\ b)\ (Var\ a))$.

[9]Technically, this statement and the preceding are not axioms but axiom schemes which describe infinite families of axioms. Because there is no convenient way to quantify over lists in first order logic, we need a separate axiom for each list of sorts, $S_1 \ldots S_n$, describing a sequence $x_1 \ldots x_n$.

Of course, we want to do more than just syntactically construct abstractions. The following three axioms provide a basis for reasoning about abstractions. The first lifts swapping.

$$\vdash \ \forall a\, b\, b' \,.\quad (b\ b') \bullet (a.\, x) = ((b\ b') \bullet a).\, ((b\ b') \bullet x) \tag{NL1}$$

To swap through an abstraction, swap the bound atom and the abstraction body. While syntacticly identical abstractions are already equal, the next axiom extends logical equality to alpha equivalent terms,

$$\vdash \ \forall a\, a'\, x\, x' \,.\quad a.\, x = a'.\, x' \iff (a = a' \wedge x = x') \vee (a' \sharp x \wedge x' = (a\ a') \bullet x). \tag{NL2}$$

Note that this corresponds to our intuitive notion of alpha equality. If the bound atoms are equal, then the bodies must be too; if the bound atoms are different, then we compare the bodies after using swap to rename occurrences of one bound atom. Lastly, we add an axiom that says we can always decompose an abstraction as follows:

$$\vdash \ \forall y : [A]S \,.\, \exists a : A \,.\, \exists x : S \,.\, y = a.\, x. \tag{NL3}$$

6.3 Case Study

Using the above framework, it is easy to define a lambda calculus theory in Nominal Logic. We will work over data sort *term* and atom sort *var*. The following signature gives the constructors in this development:

$$\begin{array}{rcl} \mathit{Var} & : & \mathit{var} \to \mathit{term} \\ \mathit{App} & : & \mathit{term} \to \mathit{term} \to \mathit{term} \\ \mathit{Lam} & : & [\mathit{var}]\mathit{term} \to \mathit{term} \end{array}$$

Because lambda abstractions are based on metalogic abstraction, this definition gives us alpha-equivalence for free. While the logic itself does not provide an induction principle for terms, Pitts adds induction as an axiom. For any predicate, P, with arity $\mathit{term}, S_1, \ldots, S_n$,

$$\begin{array}{l} \vdash \ \forall a\, x_1 \ldots x_n \,. \\ \quad P(\mathit{Var}\ a, x_1, \ldots, x_n) \ \wedge \\ \quad \forall t\, u \,.\, P(t, x_1, \ldots, x_n) \wedge P(u, x_1, \ldots, x_n) \implies P(\mathit{App}\ t\ u, x_1, \ldots, x_n) \ \wedge \\ \quad \exists a \,.\, \forall t \,.\, a \sharp t \wedge a \sharp x_1 \wedge \ldots \wedge a \sharp x_n \wedge P(t, x_1, \ldots, x_n) \implies P(\mathit{Lam}\ (a.\, t), x_1, \ldots, x_n) \\ \implies \forall t \,.\, P(t, x_1, \ldots, x_n). \end{array}$$

This is a strong induction principle. In the lambda case, we must only prove the predicate for one particular fresh variable. Thus, this induction principle is as expressive as Gordon and Melham's. Additionally, Urban and Norrish have derived similar principles for rule induction [UN05].

Pitts treats substitution by defining the following relation:

$$[\_ := \_]\_ \mapsto \_ \quad : \quad \text{relation over } (\mathit{term}, \mathit{var}, \mathit{term}, \mathit{term})$$

We will read $[a := s]t \mapsto t'$ as substituting s for a in t yields t'. One axiom is sufficient to specify the relation.

$$\begin{array}{lll} \vdash & [a := s]t \mapsto t' & \Longleftrightarrow \\ & t = \mathit{Var}\ a \wedge t' = s & \\ \vee & \exists a' \,.\, t = \mathit{Var}\ a' \wedge a' \neq a \wedge t' = \mathit{Var}\ a' & \\ \vee & \exists t_1\, t_1'\, t_2\, t_2' \,.\, t = \mathit{App}\ t_1\ t_2 \wedge t' = \mathit{App}\ t_1'\ t_2' \wedge [a := s]t_1 \mapsto t_1' \wedge [a := s]t_2 \mapsto t_2' & \\ \vee & \exists a'\, t_1\, t_1' \,.\, t = \mathit{Lam}\ b.t_1 \wedge t' = \mathit{Lam}\ b.t_1' \wedge b \sharp s \wedge [a := s]t_1 \mapsto t_1' & \end{array}$$

While it appears formidable, this axiom is just an encoding of Barendregt's partial substitution (section 2.2). The disjunction's first clause, for example, specifies that $[a := s]\,\mathit{Var}\ a \mapsto s$. Its last clause defines substitution for cases when capture will not be an issue, but is silent (as $b \sharp s$ is false) when capture could occur. Although its definition appears partial, Pitts proves that the graph of $[\_ := \_]\_ \mapsto \_$ corresponds to a total function. This definition is similar to common pencil-and-paper practice [Bar84, Pie02].

With a total definition of substitution, we can define beta reduction easily by

$$\frac{[a := s]t \mapsto t'}{\mathit{App}\ (\mathit{Lam}\ a.t)\ s \rightarrowtail_\beta t'}.$$

6.4 Evaluation

Nominal Logic provides a concrete first order environment for defining theories about binding. Because bound variables are named, theorists can write proofs about particular abstractions naturally. As predicates are equivariant, such proofs then tell us about all alpha equivalent objects. This follows standard convention for pencil-and-paper proofs. While Nominal Logic introduces a new formalization of freshness and swapping, these concepts are similar to our informal notions of free variables and renaming and should not surprise non-specialists. Additionally, being concrete, propositions in this setting are easy to read.

There have been two major attempts to mechanize Nominal Logic. In his thesis Gabbay defined a system similar to Nominal Logic as a new logic in Isabelle called Isabelle/FM. Although he was able to formalize lambda calculus in this logic, the Isabelle/FM project was never fully completed [Gab01]. Urban, with Tasson and Berghofer, is currently developing a nominal datatype package for Isabelle/HOL. However, because he is working in Isabelle/HOL (which includes Hilbert choice [NPW02]), Urban's treatment differs in several ways from pure Nominal Logical. For example, predicates must be individually proved equivariant—this property is no longer free. A major benefit of Urban's package is that consistent induction principles can be automatically derived [UT05, UB06].

Nominal Logic alone is only moderately expressive. Alpha equivalence over nominal abstractions (axiom NL2) closely tracks the Gordon-Melham alpha equivalence axiom (GM3). Otherwise the correspondence is less clear. The is-fresh-for relation is an analog of Gordon and Melham's free variable function. Likewise, Pitts's axiom NL3 is similar in spirit to Gordon and Melham's abstraction axiom (GM5). However, in either case, the precise technical connection is elusive. While it is possible to axiomatize substitution, iteration, reduction, and induction, the logic itself provides little guidance. Typical care must be taken to ensure new axioms are consistent. More generally, first order logic—on which Nominal Logic is based—is a weak logic compared with higher order logic

[GLM97]. Additionally, Pitts's definition of Nominal Logic is not compatible with the set-theoretic axiom of choice [Sup72]; this may be a serious impediment in some settings.

Nominal Logic	
Expressiveness	**B-**
Mechanizability	**B**
Transparency	**A**

7 Comparison

7.1 Expressiveness

In this paper, we evaluated the expressiveness of Nominal Logic, HOAS, and the locally named representation, using the Gordon-Melham axioms as concrete basis for comparison. The locally named representation suffers because alpha-equivalence is defined apart from real, metalogical equality. Thus all reasoning about alpha-equivalence must be done explicitly. Nominal Logic was hampered by its first order origins and inconsistency with choice. In contrast, HOAS is consistent with the the Gordon-Melham axioms, lacks expressiveness problems, and is compatible with higher order logic.

In short: HOAS $>$ Nominal Logic $\simeq$ locally named.

7.2 Mechanizability

This paper was motivated by a desire to mechanize proofs. Which model is most is the most mechanizable? Nominal logic is the youngest of our models, and has only been mechanized on an experimental basis. Gabbay's Isabelle/FM was a qualified success and Urban's looks promising. This bodes well, and there are no anticipated problems. In contrast, both the locally named and HOAS models have been mechanized. McKinna and Pollack's mechanization of local named was straightforward and successful. Automating HOAS is quite difficult, and, as we have seen, the results are not always satisfactory.

In short: locally named $>$ HOAS $\simeq$ Nominal Logic.

7.3 Transparency

Because Nominal Logic allows for reasoning about alpha equivalence classes with representative class members, it is highly transparent. The locally named representation is concrete and easy to use, but reasoning about alpha-equivalence classes must be done explicitly. In contrast, HOAS bears little resemblance to standard notation and is notoriously difficult for non-experts to understand.

In short: Nominal Logic $>$ locally named $>$ HOAS.

8 Conclusion

Nominal Logic, higher order abstract syntax, and the locally named representation can be used to reason formally about variable binding. This paper has described and evaluated each. While higher order abstract syntax is most expressive, the locally named model is most mechanizable, and Nominal Logic is most transparent to the human user. Thus, there is no one-size-fits-all formal

approach to variable binding. Rather, the needs of each formal project should be considered before choosing a binding model.

Acknowledgments

Thank you to Steve Zdancewic, Jean Gallier, and Stephanie Weirich for serving as my WPE-II committee. Additionally, I am grateful to Aaron Bohannon, Kuzman Ganchev, and Jennifer Wortman for their helpful suggestions regarding the presentation of this document.

References

[AB06] Brian Aydemir and Aaron Bohannon. Nominal techniques in Coq. To be submitted to LFMTP'06., 2006.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Tom Melham, editors, *The Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65, Oxford, United Kingdom, August 2005.

[ACM02] Simon J. Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In V. A Carreno, C. A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 13–30, Hampton, Virginia, August 2002.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 13 – 28, 1993.

[Alt02] Thorsten Altenkirch. α-conversion is easy. Under revision with draft available from `http://www.cs.nott.ac.uk/~txa/publ/alpha-draft.pdf`, 2002.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[Che05] James Cheney. Toward a general theory of names: binding and scope. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 33–40, New York, NY, USA, 2005. ACM Press.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, Cambridge, UK, 1991.

[Dav93] Martin Davis. First order logic. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Logical Foundations*, volume 1 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 31–65. Oxford University Press, 1993.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972. Also published in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.

[DFH95] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *TLCA '95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, London, UK, 1995. Springer-Verlag.

[Gab01] Murdoch J. Gabbay. *A Theory of Inductive Definitions with Alpha-Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, Cambridge University, August 2001.

[GLM97] Jean Goubault-Larrecq and Ian Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series*. Kluwer Academic, Dordrecht, Netherlands, 1997.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[GM97] Andrew D. Gordon and Thomas Melham. Five axioms of alpha-conversion. In *Ninth International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125, pages 173–190, Turku, Finland, 1997. Springer-Verlag.

[HC05] Robert Harper and Karl Crary. How to believe a Twelf proof. Available from `http://www.cs.cmu.edu/~rwh/papers/how/believe-twelf.pdf`, May 2005.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, March 1978.

[HL06] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. Draft under consideration for the Journal of Functional Programming., 2006.

[HOL05] *The HOL System: Description*, September 2005. Available from `http://hol.sourceforge.net/documentation.html`.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calclus. *Theoretical Computer Science*, 1:27–57, 1975.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 2007 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, Nice, France, January 2007. To appear.

[Lei94] Daniel Leivant. Higher order logic. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Deduction Methodologies*, volume 2 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 229–321. Oxford University Press, 1994.

[MAC01] A. Momigliano, S. J. Ambler, and R. L. Crole. A comparison of formalizations of the meta-theory of a language with variable binding in Isabelle. Technical Report EDI-INF-RR-0046, Informatics Research, University of Edinburgh, 2001.

[Mit96] John C. Mitchell. *Foundations For Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.

[MM04] Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 1–9, Snowbird, Utah, USA, 2004. ACM Press.

[MO95] Michael W. Mislove and Frank J. Oles. Full abstraction and recursion. *Theoretical Computer Science*, 151(1):207–256, 1995.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, November 1999.

[Nor03] Michael Norrish. Mechanising Hankin and Barendregt using the Gordon-Melham axioms. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding MERLIN'03*, pages 1–7, Uppsala, Sweden, August 2003. ACM Press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In R. L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[Per82] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982.

[Pfe] Frank Pfenning. Computation and deduction. In preparation with Cambridge University Press. April 1997 draft available electronically from `http://www.cs.cmu.edu/~twelf/notes/cd.ps`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[Pit03] Andrew M. Pitts. Nominal Logic, A first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Pot06] François Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148(2) of *Electronic Notes in Theoretical Computer Science*, pages 27–52, March 2006.

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. A preliminary extended version of this chapter is available from `http://cristal.inria.fr/attapl/`.

[PS99] Frank Pfenning and Carsten Schúrmann. System description: Twelf – a meta-logical framework for deductive systems. In H. Ganzinger, editor, *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.

[PS02] Frank Pfenning and Carsten Schúrmann. *Twelf User's Guide: Version 1.4*, 2002. Avaiable from `http://www.cs.cmu.edu/~twelf/`.

[Shi05] Mark R. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, University of Cambridge, February 2005.

[SP05] Mark R. Shinwell and Andrew M. Pitts. Fresh Objective Caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge, Februay 2005.

[Sup72] Patrick Suppes. *Axiomatic Set Theory*. Dover, New York, Dover edition, 1972.

[UB06] Christian Urban and Stefan Berghofer. *Nominal Datatypes in Isabelle/HOL*, March 2006. Available from `http://isabelle.in.tum.de/nominal/`.

[UN05] Christian Urban and Michael Norrish. A formal treatment of the Barendregt Variable Convention in rule inductions. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 25–32, New York, NY, USA, 2005. ACM Press.

[UT05] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *CADE-20: Proceedings of the 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, Tallinn, Estonia, July 2005. Springer.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, Massachusetts, 1993.