Engineering Formal Metatheory

Brian Aydemir
University of Pennsylvania
baydemir@cis.upenn.edu

Arthur Charguéraud
INRIA
arthur.chargueraud@inria.fr

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Randy Pollack
University of Edinburgh
rpollack@inf.ed.ac.uk

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

Abstract

Machine-checked proofs of properties of programming languages have become a critical need, both for increased confidence in large and complex designs and as a foundation for technologies such as proof-carrying code. However, constructing these proofs remains a black art, involving many choices in the formulation of definitions and theorems that make a huge cumulative difference in the difficulty of carrying out large formal developments. The representation and manipulation of terms with variable binding is a key issue.

We propose a novel style for formalizing metatheory, combining *locally nameless* representation of terms and cofinite quantification of free variable names in inductive definitions of relations on terms (typing, reduction, ...). The key technical insight is that our use of cofinite quantification obviates the need for reasoning about equivariance (the fact that free names can be renamed in derivations); in particular, the structural induction principles of relations defined using cofinite quantification are strong enough for metatheoretic reasoning, and need not be explicitly strengthened. Strong inversion principles follow (automatically, in Coq) from the induction principles. Although many of the underlying ingredients of our technique have been used before, their combination here yields a significant improvement over existing methodology, leading to developments that are faithful to informal practice, yet require no external tool support and little infrastructure within the proof assistant.

We have carried out several large developments in this style using the Coq proof assistant and have made them publicly available. Our developments include type soundness for System $F_{<:}$ and ML (with references, exceptions, datatypes, recursion and patterns) and subject reduction for the Calculus of Constructions. Not only do these developments demonstrate the comprehensiveness of our approach; they have also been optimized for clarity and robustness, making them good templates for future extension.

POPL date, City.

1. Introduction

Recent years have seen burgeoning interest in the use of proof assistants for formalizing definitions of programming languages and checking proofs of their properties. However, despite several successful *tours de force* (Appel 2001; Crary 2003; Klein and Nipkow 2006; Leroy 2006; Lee et al. 2007, etc.), the community remains fragmented, with little synergy between groups and, for newcomers wanting to join the game, a perplexing array of choices between different logics, proof assistants, and representation techniques.

To stimulate progress on this problem, Aydemir et al. (2005) proposed the POPLMARK challenge, a set of tasks designed to stress many of the critical issues in formalizing programming language metatheory. They laid out three criteria for evaluating proposed formalization techniques: *infrastructure overhead*—formalization overheads, such as additional operations and their associated proof obligations, should not be prohibitive for large developments; *transparency*—formal definitions and theorems should not depart radically from the usual informal conventions familiar to a technical audience; and *cost of entry*—the infrastructure should be usable by someone who is not an expert in theorem prover technology.

In this paper, we propose a new style for formalizing programming language metatheory that performs well on all these criteria. The style builds upon two key ingredients: locally nameless representation of syntax involving binders and a cofinite style of quantification for introducing free names in rules dealing with binders.

In locally nameless representation, bound variables are represented by de Bruijn indices while free variables are represented by names. This mixed representation combines the benefits of both approaches, avoiding difficulties associated with alpha-conversion by ensuring that each alpha-equivalence class of terms has a unique representation, while supporting formal reasoning that closely follows informal practice.

The second ingredient is the use of a cofinite quantification of free names introduced by rules dealing with binders, instead of the standard "exists-fresh" quantification—these styles are illustrated by the following variants of the typing rule for abstraction in the simply-typed lambda calculus ($\lambda_{\to}$):

$$\text{EXISTS-FRESH}\quad \frac{x \notin \mathsf{FV}(t) \quad E, x{:}S \vdash t^x \ : \ T}{E \vdash \mathsf{abs}\ t \ : \ S \to T} \qquad \text{COFINITE}\quad \frac{\forall x \notin L, \ \ E, x{:}S \vdash t^x \ : \ T}{E \vdash \mathsf{abs}\ t \ : \ S \to T}$$

Here, t^x is the body of abstraction $(\mathsf{abs}\ t)$ *opened up* with a free variable named x; in the second rule, L is some finite set of names that is chosen when we apply the rule. Just as EXISTS-FRESH is applicable if there exists some $x \notin \mathsf{FV}(t)$ such that

$E, x{:}S \vdash t^x : T$, so COFINITE is applicable if there exists some L such that $\forall x \notin L$, $E, x{:}S \vdash t^x : T$. The induction hypothesis for rule EXISTS-FRESH holds for one particular name x, which (notionally) comes from the derivation being eliminated by the induction; if that x is not fresh enough, the only way to recover is to reason about equivariance of the typing judgement. The cofinite rule, on the other hand, asserts from the start that for $E \vdash \mathsf{abs}\, t : S \to T$ to hold there must be infinitely many xs with $E, x{:}S \vdash t^x : T$; surely one of them is fresh enough, since only finitely many names can have been used so far. In rare cases, renaming is necessary for proofs, but it is straightforwardly supported by the cofinite presentation; the required lemmas may be easily derived from standard properties such as substitution and weakening, which are themselves derivable using only the cofinite definition. These renaming lemmas also provide the core arguments for the equivalence between the exists-fresh and cofinite presentations of the relations.

Neither of these ingredients is entirely new. The locally nameless representation dates back to the introduction of de Bruijn indices. Several strengthened induction principles that avoid manual renaming have been proposed (Gordon 1994; McKinna and Pollack 1993, 1999; Urban et al. 2007b). The idea of reasoning about the freshness of names by considering all but those in some finite set is at the heart of nominal logic (Gabbay and Pitts 2002) and also appears in definitions of alpha-equivalence by Krivine (1990) and Ford and Mason (2001). Our contribution lies in the precise way that we combine and apply these ingredients. In particular, the cofinitely quantified typing rule for abstraction (presented earlier), which reflects both key aspects of our design, does not appear in prior work. The observation that we can use this formulation to derive renaming lemmas with little infrastructure is a key point, of both theoretical and practical interest.

To demonstrate the comprehensiveness of our approach we have developed several significant examples in the Coq proof assistant (Coq Development Team 2007): proofs of type soundness for $\lambda_{\to}$, System $F_{<:}$ (the core of the POPLMARK challenge—parts 1A and 2A), and ML extended with references, exceptions, datatypes, recursion and patterns, as well as a proof of subject reduction for the Calculus of Constructions (including a proof of the Church-Rosser property). These developments require no external tool support and could be reproduced with another general purpose theorem prover. Little infrastructure is required between the statement of the system formalized and the core of the formal reasoning. Furthermore, our formal reasoning faithfully follows informal practice, preserving the skeletons of proofs and the kinds of arguments involved, unpolluted by details about alpha-conversion or freshness. We have carefully organized and commented these proof scripts and made them freely available as starting points for comparison and extension.[1]

The paper is organized as follows. Section 2 surveys possible approaches to representing binding structures, locates the locally nameless approach in this space, and sketches its history. Section 3 presents a complete specification of $\lambda_{\to}$ using locally nameless representation and the standard exists-fresh presentation of the semantics. Section 4, the technical meat of the paper, describes the problem faced by the exists-fresh specification, shows how cofinite quantification solves this problem, and discusses why our solution is significantly better than known alternatives. Finally, Section 5 gives a practical overview of our larger formalizations. This section also quantitatively compares our developments to each other and to other publicly available Coq solutions to the POPLMARK challenge.

[1] `http://arthur.chargueraud.org/research/2007/binders/`

2. A Survey of Binding Representations

There are two main categories of approaches to representing and reasoning about syntax with binding, depending on whether variables and binders are represented as concrete nodes in first-order algebraic structures or whether these aspects are "lifted to the metalanguage" by representing the bodies of binding constructs as metalanguage functions. In each category there are many ingenious variations, and we do not attempt to be exhaustive in this survey. Outside the main categories, there is a plethora of hybrid representations, multi-level representations, ..., about which we are silent.

2.1 Concrete approaches

With concrete (or first-order) approaches, variables are typically encoded using names or natural numbers. Capture-avoiding substitution must then be defined explicitly as a function on terms, and in the case where bound variables are named, alpha-equivalence must also be defined explicitly. Concrete approaches can be subdivided roughly in three categories: those using names to represent variables, those using de Bruijn indices, and those distinguishing bound and free variables (which may use either names or indices for each kind of variable).

The most standard representation on paper uses names to represent variables, generally quotienting raw expressions by some notion of alpha-equivalence. Although used since the first days of the lambda calculus, this representation has not proved convenient for formal developments. For example, naive substitution is not structurally recursive in this setting but requires well-founded recursion. However, the Church-Rosser property in pure lambda calculus has been proved using named representations by Homeier (2001) using HOL, Ford and Mason (2001) using PVS, and Vestergaard and Brotherston (2003) using Isabelle/HOL (unusually, Vestergaard and Brotherston reason about unquotiented raw terms). Other attempts to tame this representation include work by Stoughton (1988), where simultaneous substitution is defined structurally and puts terms into a canonical alpha-normal form, and by Hendriks and van Oostrom (2003).

To overcome the clumsiness of explicit alpha-conversion or quotienting, the first large formalizations of languages with binders were based on de Bruijn indices (1972) rather than names. In this approach, each alpha-equivalence class of lambda-terms has a unique syntactic representation in which variables are encoded using natural numbers giving the depth of the variable relative to its binder. Indices greater than the number of enclosing local binders may indicate a position in some enclosing context such as a typing context. There is abundant evidence of the effectiveness of this representation, from proofs of the Church-Rosser property for lambda calculus—for example, by Shankar (1988) using the Boyer-Moore prover, by Huet (1994) in Coq, by Rasmussen (1995) in Isabelle/ZF, and by Nipkow (2001) in Isabelle/HOL—to harder results such as strong normalization for System F in LEGO (Altenkirch 1993) and formalizing Coq in Coq (Barras and Werner 1997). In de Bruijn representation, the treatment of bound variables incurs minor technical annoyances (lifting over binders, etc.), but the treatment of free variables (e.g., variables bound by a typing context) requires reasoning far from natural informal presentations. For example, consider the statement of a lemma that says thinning the typing context (weakening and permuting) preserves a judgement: indices occurring in the judgement must be updated to match the permutation of their binding points in the context.

Nominal logic (Gabbay and Pitts 2002; Pitts 2003) provides another way to address the problem of alpha-conversion inherent to the named representation. Urban (2007) has adapted these ideas as a library in a standard higher-order logic, and Urban et al. (2007a) are developing in Isabelle/HOL a *nominal package* that provides facilities for formal reasoning with binders over names. The pack-

age helps by automating language definitions that equate alpha-equivalent terms, by providing lemmas about renaming and freshness of names, by deriving strengthened induction principles that provide name freshness facts for relations over these languages, and by supporting recursion over these languages. Relations and functions defined over such languages must be shown to be equivariant (stable under renaming of free variables) and the package includes automation that states and solves all such goals within a useful syntactic class. There are some restrictions that must be observed to use this package. All nominal structures must be finitely supported, the rules of inductive definitions must meet certain requirements, and the representation is not canonical, so alpha-equivalence shows through to the user in places (e.g., $\lambda x.s = \lambda y.t$ does not imply that $x = y$ and $s = t$). The inversion principles automatically derived from inductively defined relations are weaker than needed for natural reasoning.

Besides these "homogeneous" concrete approaches—where there is a single notion of "variable," either a name or a de Bruijn index—there is a third category of concrete representations that uses two distinct syntactic classes, called variables (for locally bound variables) and parameters (for free, or globally bound variables). The idea that variables and parameters should be distinguished goes back at least to Gentzen and Prawitz. Several combinations can be considered, using names, indices, or levels to represent bound and free variables; of these, two have been used in the literature.

The first is the *locally named* representation, which uses distinct species of names to represent variables and parameters respectively. McKinna and Pollack introduced this technique to formalize Pure Type Systems (PTS) (1993; 1999), following a suggestion by Coquand (1991). This representation avoids the difficulties of reasoning about capture-avoiding substitutions: since variables and parameters are syntactically distinguished, no parameter can ever be captured by a variable binder during substitution. McKinna and Pollack also introduced a technique for handling the requirement of choosing fresh global variables that often occurs in reasoning about binding (weakening lemmas are a prototypical example of the problem). By careful choice of definitions, they avoid the need to reason about alpha-equivalence of bound variables throughout their large formal body of PTS metatheory. With these techniques, reasoning about lambda calculus and type theory is straightforward, if heavy.

Nonetheless, the use of names for bound variables is not a perfect fit to the intuitive notion of binding: eventually one needs to reason about alpha-conversion. For example, parallel reduction has the diamond property concretely in locally named representation, but beta reduction has this property only up to alpha-conversion (Pollack 1994b, Section 3.3.6). So Pollack (1994a) suggested that the McKinna–Pollack approach to reasoning with two species of variables also works well with a representation that uses names for parameters and de Bruijn indices for bound variables. This *locally nameless* representation, in which alpha-equivalence classes have canonical representation, was already mentioned in the conclusion of de Bruijn's famous paper (1972). It had been used for implementation in Huet's *Constructive Engine* (1989), and later in the implementations of the Coq, LEGO (Luo and Pollack 1992), HOL 4 (Norrish and Slind 2007), and EPIGRAM (McBride and McKinna 2004) proof assistants.

In the context of formal proofs, Gordon (1994) appears to be the first to have used locally nameless representation. Rather than reason with locally nameless terms directly, he builds a representation of named lambda terms on top of locally nameless terms and demonstrates how the named terms may be used as a basis for representing syntax. Using this technique, Gordon formalized Abramsky's lazy lambda calculus. Later work by Gordon and Melham (1996) also uses locally nameless terms as a model for an abstract "axiomatic" representation of named terms. Pollack (2006) has more recently emphasized the benefits of locally nameless representation in the context of the POPLMARK challenge. Locally nameless representation with (variants of) McKinna–Pollack style reasoning has been used by several researchers (with several proof tools) for solutions to the POPLMARK challenge, first by Leroy (2007), and later by Chlipala (2006), and Ricciotti (2007).

2.2 Higher-order approaches

Higher-order representations use the function space of the meta-logic to encode binding in an object language, allowing issues like capture-avoidance and alpha-equivalence to be handled once and for all by the meta-logic, rather than facing them anew for each object language. There is a bewildering variety of higher-order approaches, which we survey only superficially.

In *higher-order abstract syntax* (HOAS) (Pfenning and Elliot 1988), the introduction form for lambda-abstractions has type `(term->term)->term`, i.e., the lambda constructor packages a function of type `(term->term)`, which should be thought of as the function substituting its argument into the body of the lambda. Not only are alpha-equivalent terms equal, but substitution is handled by the meta-language. Due to the negative occurrence of `term` in this type, there is no inductive datatype of terms in this encoding. HOAS was introduced by Church and developed by de Bruijn, Martin-Löf, Plotkin, Pfenning, and their co-workers. The modern form, using dependent types and *hypothetical* and *schematic* judgements as well as HOAS term representation, appeared in the Edinburgh Logical Framework (ELF) (Harper et al. 1993). By design, ELF is a weak type theory to be used as a metalanguage, not supporting inductive definition or primitive recursion. It allows *faithful* representation of object languages, where ELF itself adds no mathematical strength to the object language representation. The use of ELF methodology for metatheory (as opposed to just representation) has been highly developed by Pfenning and his co-workers. The implementation of this approach is the Twelf system (Pfenning and Schürmann 1999), which is widely used and very successful for formalizing the metatheory of a variety of programming languages (Ashley-Rollman et al. 2005; Lee et al. 2007). The approach continues to be developed foundationally as well as in practice (Harper and Licata 2007).

A second main stream of work, *weak* higher-order encodings, gives a type such as `(name->term)->term` to the lambda constructor of terms. This constructor packages a function of type `(name->term)`, which should be thought of as a function that takes a name and returns the instance of the lambda's body instantiated with that name. Here, alpha-equivalent terms are equated, but substitution must be defined as a relation between terms. Despeyroux et al. (1995) demonstrate an early use of this approach. The approach has been most developed by Honsell, Miculan, and their co-workers as the Theory of Contexts (Honsell et al. 2001; Bucalo et al. 2006).

2.3 Why Locally Nameless?

We have chosen to concentrate on concrete approaches to represent binding structure. Although higher-order approaches are attractive to use and show promise in many large examples, they are, at present, limited in scope. Twelf is the only game in town for metatheory about HOAS representations, and it is (at present) strictly limited by proof theoretic strength. The Theory of Contexts is also interesting, but hasn't been taken up outwith its originators. These are the subject of active research and may become more generally applicable in the future.

Among the concrete approaches, locally nameless representation offers the best combination of features. Because free variables are represented using names, lemmas are stated and proofs struc-

tured just as they are with the standard representation using names, making locally nameless developments intuitive. In particular, the shifting operations that clutter both statements and proofs in a pure de Bruijn setting are avoided. Because bound variables are represented using indices, each alpha-equivalence class of lambda terms has a unique representation, thus avoiding the difficulties associated with alpha-equivalence. Finally, because bound and free variables are syntactically distinguished there is no issue of variable capture, and substitution can be defined by simple structural recursion.

These observations motivate the choice of the locally nameless representation. The next section presents this style in detail.

3. Locally Nameless Representation

In this section, we describe locally nameless representation, using $\lambda_{\to}$ as a running example. All the material presented is formal, in the sense that we have implemented it in Coq.[2] We write $\Rightarrow$ for logical implication.

3.1 Definitions

The fundamental idea of the locally nameless approach is to distinguish bound from free variables, using de Bruijn indices for the former and names for the latter. Figure 1 presents a locally nameless specification of $\lambda_{\to}$. An occurrence of a bound variable is represented as $(\mathsf{bvar}\ i)$ where i is a natural number index denoting the number of enclosing lambda abstractions that must be traversed before reaching the abstraction binding that variable. Abstractions do not bind names; for example, $(\mathsf{abs}\ (\mathsf{bvar}\ 0))$ is the identity function.

In its raw form, this representation is not isomorphic to ordinary named lambda terms because a bound variable may not resolve to any binder. For example, $(\mathsf{abs}\ (\mathsf{bvar}\ 2))$ does not represent any lambda-term. We therefore define a predicate, $(\mathsf{term}\ t)$, on *pre-terms* (the syntactic objects defined by the grammar in Figure 1) that selects the *locally closed* pre-terms—or just *terms*—in which all indices resolve to binders.

The key operation on pre-terms is *opening* an abstraction by instantiating a bound variable with a term. If $(\mathsf{abs}\ t)$ is an abstraction and u another term, then the result of opening the abstraction with u, written t^u, is formed by replacing in the body t all bound variables that point to the outermost lambda with u. Figure 1 gives the inductive implementation of the open function; the idea is to explore the term, maintaining the current number of binders passed through.[3] One use of the open operation is in the conclusion of the beta reduction rule, RED-BETA, near the bottom of the figure.

The open operation is also frequently applied to a free variable; we call this specialized version *variable opening*, which we write with a slight abuse of notation as (t^x) instead of $(t^{\mathsf{fvar}\ x})$. Variable opening is used when passing through a binder to turn the bound variable into a free variable. In a named representation, we may have our hands on an abstraction $(\mathsf{abs}\ x\ t)$ and wish to talk about its body t. With the locally nameless representation, the abstraction would be written $(\mathsf{abs}\ t)$; to talk about its body as a term (rather than a pre-term with an unbound index), we open t with some fresh name x. A typical example of variable opening appears in the typing rule for abstractions, TYPING-ABS.

In variable opening, the chosen variable should not already appear free inside the term. Otherwise, the operation would be a form of variable capture: the name being given to the index, a bound variable, would be the same as a free variable. Therefore,

[2] The corresponding development is contained in files `STLC_Core_*.v`.

[3] Experts will note that this definition of open works correctly only when zero is the only unbound index. The definition can be generalized to allow multiple binders to be opened simultaneously as explained in Section 5 in the paragraph about our ML development.

Syntax:

$$\begin{array}{lcl} S,T & \equiv & A \mid T_1 \to T_2 \\ t,u,w & \equiv & \mathsf{bvar}\ i \mid \mathsf{fvar}\ x \mid \mathsf{app}\ t_1\ t_2 \mid \mathsf{abs}\ t \\ E,F,G & \equiv & \varnothing \mid E, x{:}T \end{array}$$

Open: $t^u \equiv \{0 \to u\}\, t$, with

$$\begin{array}{lcl} \{k \to u\}\,(\mathsf{bvar}\ k) & = & u \\ \{k \to u\}\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \quad \text{when } i \neq k \\ \{k \to u\}\,(\mathsf{fvar}\ x) & = & \mathsf{fvar}\ x \\ \{k \to u\}\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\,(\{k \to u\}\,t_1)\ (\{k \to u\}\,t_2) \\ \{k \to u\}\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ (\{(k+1) \to u\}\,t) \end{array}$$

Free variables:

$$\begin{array}{lcl} \mathsf{FV}(\mathsf{bvar}\ i) & = & \varnothing \\ \mathsf{FV}(\mathsf{fvar}\ x) & = & \{x\} \\ \mathsf{FV}(\mathsf{app}\ t_1\ t_2) & = & \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \\ \mathsf{FV}(\mathsf{abs}\ t) & = & \mathsf{FV}(t) \end{array}$$

Locally closed terms:

$$\frac{}{\mathsf{term}\ (\mathsf{fvar}\ x)}\ \textsc{term-var} \qquad \frac{\mathsf{term}\ t_1 \qquad \mathsf{term}\ t_2}{\mathsf{term}\ (\mathsf{app}\ t_1\ t_2)}\ \textsc{term-app}$$

$$\frac{x \notin \mathsf{FV}(t) \qquad \mathsf{term}\ (t^x)}{\mathsf{term}\ (\mathsf{abs}\ t)}\ \textsc{term-abs}$$

Well-formed environments (no duplicate names):

$$\frac{}{\mathsf{ok}\ \varnothing}\ \textsc{ok-nil} \qquad \frac{\mathsf{ok}\ E \qquad x \notin \mathsf{dom}(E)}{\mathsf{ok}\ (E, x{:}T)}\ \textsc{ok-cons}$$

Typing:

$$\frac{\mathsf{ok}\ E \qquad (x{:}T) \in E}{E \vdash \mathsf{fvar}\ x\ :\ T}\ \textsc{typing-var}$$

$$\frac{E \vdash t_1\ :\ S \to T \qquad E \vdash t_2\ :\ S}{E \vdash \mathsf{app}\ t_1\ t_2\ :\ T}\ \textsc{typing-app}$$

$$\frac{x \notin \mathsf{FV}(t) \qquad E, x{:}T_1 \vdash t^x\ :\ T_2}{E \vdash \mathsf{abs}\ t\ :\ T_1 \to T_2}\ \textsc{typing-abs}$$

Call-by-value evaluation:

$$\frac{\mathsf{term}\ (\mathsf{abs}\ t)}{\mathsf{value}\ (\mathsf{abs}\ t)}\ \textsc{value-abs}$$

$$\frac{\mathsf{term}\ (\mathsf{abs}\ t) \qquad \mathsf{value}\ u}{\mathsf{app}\ (\mathsf{abs}\ t)\ u \longmapsto t^u}\ \textsc{red-beta}$$

$$\frac{t_1 \longmapsto t_1' \qquad \mathsf{term}\ t_2}{\mathsf{app}\ t_1\ t_2 \longmapsto \mathsf{app}\ t_1'\ t_2}\ \textsc{red-app-1}$$

$$\frac{\mathsf{value}\ t_1 \qquad t_2 \longmapsto t_2'}{\mathsf{app}\ t_1\ t_2 \longmapsto \mathsf{app}\ t_1\ t_2'}\ \textsc{red-app-2}$$

Type soundness lemmas (preservation and progress):

$$E \vdash t\ :\ T \quad \Rightarrow \quad t \longmapsto t' \quad \Rightarrow \quad E \vdash t'\ :\ T$$

$$\varnothing \vdash t\ :\ T \quad \Rightarrow \quad (\mathsf{value}\ t \ \vee \ \exists t',\ t \longmapsto t')$$

Figure 1. Locally nameless presentation of $\lambda_{\to}$

Substitution of a term for a free name:

$$
\begin{array}{lcl}
[z \to u]\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \\
[z \to u]\,(\mathsf{fvar}\ z) & = & u \\
[z \to u]\,(\mathsf{fvar}\ x) & = & \mathsf{fvar}\ x \quad \text{when } x \neq z \\
[z \to u]\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\ ([z \to u]\,t_1)\ ([z \to u]\,t_2) \\
[z \to u]\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ ([z \to u]\,t)
\end{array}
$$

Properties of substitution:

$$
\begin{array}{ll}
\mathsf{subst\_fresh} & x \notin \mathsf{FV}(t) \ \Rightarrow\ [x \to u]\,t = t \\
\mathsf{subst\_open} & [x \to u]\,(t^w) = ([x \to u]\,t)^{([x \to u]\,w)} \\
 & \text{when } \mathsf{term}\ u \\
\mathsf{subst\_open\_var} & [x \to u]\,(t^y) = ([x \to u]\,t)^y \\
 & \text{when } x \neq y \text{ and } \mathsf{term}\ u \\
\mathsf{subst\_intro} & [x \to u]\,(t^x) = t^u \\
 & \text{when } x \notin \mathsf{FV}(t)
\end{array}
$$

Figure 2. Substitution

the rule TYPING-ABS also includes the premise $x \notin \mathsf{FV}(t)$, which uses the *free variable* function to collect the set of free variables in a pre-term. Like open, this function is defined by simple structural recursion and need not worry about confusing free and bound variables—*all* variable names in the term are free, since bound variables are represented by indices.

To state the typing judgement of $\lambda_{\to}$, we need to formalize typing environments. In Figure 1, environments are concretely represented as association lists, for which we assume some standard operations. The *concatenation* of two environments is written $(E,\ F)$. The *lookup* of a variable in an environment is expressed by the predicate $(x{:}T) \in E$, which holds when (x, T) is the rightmost pair of E whose first component is x. Finally, a variable is *fresh* for an environment, written $x \notin \mathsf{dom}(E)$, when that variable does not appear in the first component of any pair in E. In principle, the association list representation allows multiple assumptions for the same variable, but our typing rules require that environments are well formed (ok) in the sense that a given variable is bound at most once in a given environment.

The final portion of Figure 1 completes the definition of $\lambda_{\to}$ by stating the properties we wish to prove: the standard preservation and progress lemmas. The figure, as a whole, forms our "trusted base" of definitions: one must believe in the correctness of everything stated in the figure in order to believe that a formal proof of type soundness is really about one's informal notion of $\lambda_{\to}$. We discuss this issue in additional detail in Section 3.4.

3.2 Substitution

The definition of $\lambda_{\to}$ depends only on the open and FV operations. In proofs, however, the operation of *free variable substitution* is also fundamental; it is defined in Figure 2. The substitution function (written $[x \to u]\,t$) replaces free names with terms. It can be defined by simple structural recursion since it does not need to be capture avoiding. As in the definition of open, there is no arbitrary choice of name for the bound variable in the abstraction case, so we need not worry that this choice will affect the result. Therefore, the behavior of substitution is natural and easy to reason about.

The properties of the substitution function, shown in the bottom portion of Figure 2, are fundamental when working with a locally nameless representation. The lemma $\mathsf{subst\_fresh}$ states that substitution for a unused variable has no effect, and the lemma $\mathsf{subst\_open}$ states that substitution distributes over the opening operation. The remaining two lemmas are immediate corollaries of the two former. The lemma $\mathsf{subst\_open\_var}$ permutes a substitution with variable opening, and the lemma $\mathsf{subst\_intro}$ decomposes an open operation into a variable opening operation and a substitution. Examples of when these lemmas are used can be found in the proof of the substitution lemma for $\lambda_{\to}$ (Section 4.1) and in the proof of a renaming lemma (Section 4.2).

Close: ${}^{\backslash x}t \equiv \{0 \leftarrow x\}\,t$, with

$$
\begin{array}{lcl}
\{k \leftarrow x\}\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \\
\{k \leftarrow x\}\,(\mathsf{fvar}\ x) & = & \mathsf{bvar}\ k \\
\{k \leftarrow x\}\,(\mathsf{fvar}\ y) & = & \mathsf{fvar}\ y \quad \text{when } x \neq y \\
\{k \leftarrow x\}\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\ (\{k \leftarrow x\}\,t_1)\ (\{k \leftarrow x\}\,t_2) \\
\{k \leftarrow x\}\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ (\{(k+1) \leftarrow x\}\,t)
\end{array}
$$

Figure 3. Variable Closing

We occasionally need a related operation—intuitively, a reciprocal of variable opening—that abstracts the name x from the term t, replacing it with a bound variable. We call this operation *variable closing* and define it in Figure 3.

3.3 Working with Local closure

Although the relations defining the semantics of $\lambda_{\to}$ in Figure 1 are formally defined over pre-terms, these relations actually include only locally closed terms and the basic operations preserve local closure.

$$
\begin{array}{lcl}
\mathsf{term}\ (\mathsf{abs}\ t) \ \wedge\ \mathsf{term}\ u & \Rightarrow & \mathsf{term}\ (t^u) \\
\mathsf{term}\ u \ \wedge\ \mathsf{term}\ t & \Rightarrow & \mathsf{term}\ ([x \to u]\,t) \\
E \vdash t\ :\ T & \Rightarrow & \mathsf{term}\ t \\
t \longmapsto t' & \Rightarrow & \mathsf{term}\ t \ \wedge\ \mathsf{term}\ t'
\end{array}
$$

We formulate the definitions in this way for two reasons. First, it is required for the adequacy of our formalization (only locally closed terms correspond to alpha-equivalence classes of ordinary named terms). And second, some useful properties of the definitions are true only for locally closed terms.[4]

To ensure the last property above, the definition of small-step evaluation includes premises about local closure in several places. In general, such premises are only required at the leaves of the definition: for example, we do not need to have a premise $(\mathsf{term}\ t)$ if there is already a premise $(\mathsf{value}\ t)$, since the latter implies the former.

Because our definitions guarantee local closure, we usually do not need to add local closure hypotheses to the statements of lemmas and theorems. For example, the "substitution preserves typing" lemma

$$
E,\ z{:}S,\ F \vdash t\ :\ T \quad \Rightarrow \quad E \vdash u\ :\ S \quad \Rightarrow \\
E,\ F \vdash [z \to u]\,t\ :\ T
$$

requires none because the typing relations ensure that t, u, and $[z \to u]\,t$ are all locally closed. As a result, even though our use of the nameless representation means that we must be careful about local closure, the management of this predicate does not require a significant departure from common informal practice.

The inductive definition of local closure that we chose is given in Figure 1. First, a bound variable on its own is never locally closed, and second, an abstraction is locally closed if the body of that abstraction, once opened up with a fresh name, is locally closed.[5] There are several ways to define local closure, but we favor this style which is similar to the `VClosed` relation of McKinna and Pollack (1993). As they point out, an advantage of this definition

[4] These include both low-level technical properties such as $\mathsf{subst\_open}$ and more interesting properties such as reflexivity of subtyping in System $F_{<:}$.

[5] The freshness assumption is actually not required for this judgement. However, in our development, we maintain the invariant that we never open a term with a variable that occurs free in that term. This streamlines proofs in many places.

is that it provides a structural induction principle for locally closed terms:

$$\frac{\begin{array}{l}\forall\, x,\ P(x) \\ \forall\, t_1,\ \forall\, t_2,\ P(t_1) \ \Rightarrow\ P(t_2) \ \Rightarrow\ P(\mathsf{app}\ t_1\ t_2) \\ \forall\, x,\ \forall\, t,\ x \notin \mathsf{FV}(t) \ \Rightarrow\ P(t^x) \ \Rightarrow\ P(\mathsf{abs}\ t)\end{array}}{\forall t,\ \mathsf{term}\ t \ \Rightarrow\ P(t)}$$

3.4 Adequacy

Why should you believe this formal representation of $\lambda_{\rightarrow}$ is "correct," i.e., expresses your understanding of the informal system? This is the question of *adequacy* of the representation, set out by Harper et al. (1993) in their context of HOAS. First, note that it is an informal question, because it involves the relationship between an informal thing and a formal thing. No matter how much faith you put in Coq, no Coq proof will completely settle this question.

Nonetheless, formal proofs are useful for this question in three ways. First, you may prove that the formalization has the properties you expect from the informal system and does not have unexpected properties. E.g., our Coq representation of $\lambda_{\rightarrow}$ is shown to have Church–Rosser, weakening, subject reduction, etc., some of which is discussed below. (You must also be convinced that our formal statements of these properties are correct.) If you think our rule TYPING-ABS requires a stronger side condition, prove your expected rule is admissible in our system. Second, you may prove some formal relationship with another formalization of the same informal system that you, and other readers, believe to be adequate, e.g., that our formalization is related to a pure de Bruijn representation in some way (but we haven't done that). Finally, if intensional, impredicative, constructive type theory is just too weird, you may carry out the same representation and its development in some other proof system.

Harper et al. (1993) ask that there be a compositional bijection between the (informal) syntactic entities (terms, formulas, proofs) and some collection of entities of the formal representation. (They also note this may be taken to have different meanings: e.g., adequacy for theorems versus adequacy for derivations.) In our setting, you can carry this out (on paper, not in Coq) with a fairly simple bijection.

4. Cofinite Quantification

Although the definitions in the previous section adequately represent $\lambda_{\rightarrow}$, they yield insufficiently strong induction principles: in many situations we must rename the variable used to open an abstraction to complete a proof. The need to reason about renaming motivates our use of "cofinite quantification" for inductively defined relations.

In this section, we present a complete style based on cofinite quantification. We first explain cofinite quantification and give a new definition of $\lambda_{\rightarrow}$ using it. Definitions in this style are sufficient, by themselves, to develop significant amounts of metatheory —e.g., to prove weakening and substitution results— since they naturally balance strong induction and inversion principles with useful introduction forms. In a few situations, however, *renaming lemmas* are required to build derivations using the cofinitely quantified rules. We show how these lemmas can be derived directly from weakening and substitution. Finally, to be confident that relations defined with cofinite quantification are adequate encodings of $\lambda_{\rightarrow}$, we use renaming to show that they are equivalent to their counterparts shown in Figure 1.

Having described our complete style of development, we then observe that some significant streamlining is possible, notably in cases where one believes in the adequacy of the cofinite presentation of a relation without requiring a formal proof of equivalence with the "exists-fresh" variant. We conclude the section with comparisons to previous work, arguing that the added value of our approach is significant in each case.

4.1 Stronger Induction Principles

The definitions of the local closure and typing relations in Figure 1 use an "exists-fresh" style of quantification. When an abstraction is opened in the premise of a rule (TERM-ABS and TYPING-ABS), the name used in the premise is required to be fresh for the body of the abstraction. Thus, to build a derivation using one of these rules, it suffices to show that there exists a single sufficiently fresh name for which the premise holds. The premises of rules written in this way are easy to satisfy, making them ideal introduction forms.

However, such rules give rise to weak elimination forms (inversion and induction principles). The fact that the premises of a rule defined using exists-fresh quantification need only hold for one particular name means that the corresponding induction hypothesis also holds for one particular name. That name will only be as fresh as the premise of the rule requires, which may not be enough.

For example, consider the proof of the weakening lemma for the typing relation, which allows one to insert additional typing assumptions anywhere into a typing environment:

$$\begin{array}{l}\mathsf{typing\_weaken}: \\ \quad E,\, G \vdash t\ :\ T \ \Rightarrow\ \mathsf{ok}\ (E,\, F,\, G) \ \Rightarrow \\ \quad E,\, F,\, G \vdash t\ :\ T\,.\end{array}$$

When proving this by induction on the given typing derivation, in the case for TYPING-ABS we are given a derivation ending with

$$\frac{x \notin \mathsf{FV}(t) \qquad E,\, G, x{:}T_1 \vdash {t_1}^x\ :\ T_2}{E,\, G \vdash \mathsf{abs}\ t_1\ :\ T_1 \rightarrow T_2}$$

and an induction hypothesis

$$\mathsf{ok}\ (E,\, F,\, G, x{:}T_1) \ \Rightarrow\ (E,\, F,\, G, x{:}T_1 \vdash {t_1}^x\ :\ T_2)$$

and we must derive

$$E,\, F,\, G \vdash \mathsf{abs}\ t_1\ :\ T_1 \rightarrow T_2$$

for an arbitrary F. By TYPING-ABS, it suffices to derive

$$E,\, F,\, G, y{:}T_1 \vdash {t_1}^y\ :\ T_2\,,$$

for some name $y \notin \mathsf{FV}(t_1)$. Intuitively, we want to use $y = x$, since the induction hypothesis applies to x. However, the given typing derivation shows that ${t_1}^x$ is well typed in the environment $(E,\, G, x{:}T_1)$, so we can only derive that x is fresh for $(E,\, G)$. We need to type ${t_1}^x$ in the extended environment $(E,\, F,\, G, x{:}T_1)$, but x is not guaranteed to be sufficiently fresh for that environment as it could appear in F.

The particular name x appearing in the induction hypothesis is (notionally) the name that occurs in that position of the particular derivation of $E,\, G \vdash t\ :\ T$ being eliminated in the proof. In an informal proof, we avoid the problem by assuming this derivation uses sufficiently fresh names. (Depending on our informal view, this is usually justified by appeal to the Barendregt Variable Convention (1984), which allows us to assume that the name of the bound variable in the abstraction case is sufficiently fresh.)

Our solution is to strengthen the induction principle for the typing relation by changing its definition to the one at the bottom of Figure 4. This definition differs only in the rule for abstractions, C-TYPING-ABS. This new rule intuitively states: "to show that $(\mathsf{abs}\ t)$ is well typed, it suffices to show that t^x is well-typed for any x not in some finite set of names, L." We say that this definition of the typing relation uses "cofinite quantification" since we must show that the premise of C-TYPING-ABS holds for all but finitely many names. The set of names L may be different for each use of the rule. To similarly strengthen the structural induction principle for terms, we redefine term in the cofinite style at the top of Figure 4,

$$\frac{}{\mathsf{term}_c\ (\mathsf{fvar}\ x)}\ \text{C-TERM-VAR}$$

$$\frac{\mathsf{term}_c\ t_1 \qquad \mathsf{term}_c\ t_2}{\mathsf{term}_c\ (\mathsf{app}\ t_1\ t_2)}\ \text{C-TERM-APP}$$

$$\frac{\forall x \notin L,\ \ \mathsf{term}_c\ (t^x)}{\mathsf{term}_c\ (\mathsf{abs}\ t)}\ \text{C-TERM-ABS}$$

$$\frac{\mathsf{ok}\ E \qquad (x{:}T) \in E}{E \vdash_c \mathsf{fvar}\ x\ :\ T}\ \text{C-TYPING-VAR}$$

$$\frac{E \vdash_c t_1\ :\ S \to T \qquad E \vdash_c t_2\ :\ S}{E \vdash_c \mathsf{app}\ t_1\ t_2\ :\ T}\ \text{C-TYPING-APP}$$

$$\frac{\forall x \notin L,\ \ (E, x{:}T_1 \vdash_c t^x\ :\ T_2)}{E \vdash_c \mathsf{abs}\ t\ :\ T_1 \to T_2}\ \text{C-TYPING-ABS}$$

Figure 4. Local closure and typing using cofinite quantification

and replace all references to term in the reduction relation with this one. Although this new definition of $\lambda_{\to}$ differs from the one given in Figure 1, it is equivalent, as we show in Section 4.3.

Using cofinite quantification in the rules provides a stronger induction principle because, in the cases that open abstractions, the induction hypotheses will hold for all names except those in some finite set L, rather than just for a single name. On the other hand, a rule defined using cofinite quantification is nearly as easy to use for introduction as the exists-fresh version because L can include the finitely many names that could potentially lead to clashes.

Consider again the proof of the weakening lemma—this time, for the cofinite typing relation $\vdash_c$. In the case for C-TYPING-ABS, we are given the assumption

$$\forall x \notin L',\ \ (E, G, x{:}T_1 \vdash_c t_1{}^x\ :\ T_2)$$

for some finite set of names L', and an induction hypothesis

$$\forall x \notin L',\ \ \mathsf{ok}\ (E, F, G, x{:}T_1) \quad \Rightarrow \\ E, F, G, x{:}T_1 \vdash_c t_1{}^x\ :\ T_2\,,$$

and we must derive

$$E, F, G \vdash_c \mathsf{abs}\ t_1\ :\ T_1 \to T_2\,.$$

By C-TYPING-ABS, it suffices to find an L for which we can show

$$\forall x \notin L,\ \ (E, F, G, x{:}T_1 \vdash_c t_1{}^x\ :\ T_2)\,.$$

Choosing $L = L' \cup \mathsf{dom}(F)$ does the trick: any $x \notin L$ is also not in L', so the induction hypothesis directly gives us the result.

As an additional example of the usefulness of the stronger induction hypothesis, we prove the standard "substitution preserves typing" lemma for the cofinite typing relation:

typing_subst :
$$E, z{:}S, F \vdash_c t\ :\ T \quad \Rightarrow \quad E \vdash_c u\ :\ S \quad \Rightarrow \\ E, F \vdash_c [z \to u]\, t\ :\ T\,.$$

The proof of this lemma also demonstrates the ease of working with substitution in a locally nameless setting. It proceeds by induction on the given typing derivation for t. In the case for C-TYPING-ABS, we have a derivation that ends in

$$\frac{\forall x \notin L',\ \ (E, z{:}S, F, x{:}T_1 \vdash_c t_1{}^x\ :\ T_2)}{E, z{:}S, F \vdash_c \mathsf{abs}\ t_1\ :\ T_1 \to T_2}$$

for some finite set of names L', and an induction hypothesis

$$\forall x \notin L',\ \ (E, F, x{:}T_1 \vdash_c [z \to u]\,(t_1{}^x)\ :\ T_2)\,,$$

and we must derive

$$E, F \vdash_c [z \to u]\,(\mathsf{abs}\ t_1)\ :\ T_1 \to T_2\,.$$

By the definition of substitution, we simplify the substitution to $\mathsf{abs}\ ([z \to u]\, t_1)$. Then to apply C-TYPING-ABS, we must find an L for which we can show

$$\forall x \notin L,\ \ (E, F, x{:}T_1 \vdash_c ([z \to u]\, t_1)^x\ :\ T_2)\,.$$

We use subst_open_var to write $([z \to u]\, t_1)^x$ as $[z \to u]\,(t_1{}^x)$ (to match the induction hypothesis). To do this rewriting, we need x and z to be distinct. Since the induction hypothesis holds only for $x \notin L'$, we choose $L = L' \cup \{z\}$.

In this proof and in the proof of weakening, we carefully chose L when applying C-TYPING-ABS to create a new typing derivation. In practice, we use a tactic to instantiate L with the set of all names appearing in the current proof context: intuitively, showing that a judgement holds for fewer names is easier than showing that it holds for more.

4.2 Renaming

There are few situations in which we need to apply a derivation rule with a cofinitely quantified premise but only know that this premise holds for one particular fresh name. Alternatively, knowing that an abstraction type checks, we may wish to use inversion to show that the body of an abstraction type checks with a particular name, but we do not know that that particular name is excluded from the set L used with that invocation of C-TYPING-ABS. In such situations, we need to explicitly invoke the fact that our judgments are equivariant through a *renaming lemma*. We present now an example of such a situation and then explain how to prove the renaming lemma by reusing the corresponding substitution lemma.

Suppose that we are trying to show that type checking is decidable. [6]

typing_decidable :
$$\mathsf{term}\ t \ \Rightarrow\ \mathsf{ok}\ E \ \Rightarrow\ (E \vdash_c t\ :\ T) \vee \neg(E \vdash_c t\ :\ T)$$

We prove this result by induction on term t. Consider the case for abstractions (where $t = \mathsf{abs}\ t_1$) with an arbitrary environment E and an arrow type (where $T = T_1 \to T_2$). Here we are given an induction hypothesis that states that type checking the body is decidable (for some finite set L').

$$\forall x \notin L',\ \ \forall E'\ T', \\ \mathsf{ok}\ E' \ \Rightarrow\ (E' \vdash_c t_1{}^x\ :\ T') \vee \neg(E' \vdash_c t_1{}^x\ :\ T')$$

At this point, we must determine whether the body type checks to know whether the abstraction should type check. So we pick an arbitrary fresh variable, $x \notin L' \cup \mathsf{FV}(t_1) \cup \mathsf{dom}(E)$, and consider the two cases that arise from instantiating E' with $(E,\ x{:}T_1)$ and T' with T_2.[7] If the body does type check, we have a derivation of

$$E, x{:}T_1 \vdash_c t_1{}^x\ :\ T_2$$

However, to show that the abstraction type checks, using rule C-TYPING-ABS, we must show that there is some L such that

$$\forall y \notin L,\ \ E, y{:}T_1 \vdash_c t_1{}^y\ :\ T_2$$

Although x was arbitrary, we do not know whether the particular variable that we chose influenced whether the body type checked.

[6] The simplicity of the example below requires a slight change to the formalization of $\lambda_{\to}$ that annotates applications (app) with the argument type. This change permits the proof to go through in the application case, but is otherwise independent of binding.

[7] The variable x must be fresh for L' for the induction hypothesis to apply and must be fresh for $\mathsf{dom}(E)$ so that $\mathsf{ok}\ (E,\ x{:}T_1)$. We also include $\mathsf{FV}(t_1)$ to maintain the invariant that terms are only opened with fresh variables.

Therefore, we complete the proof with the help of a renaming lemma that asserts that typing judgments are stable under certain renamings:[8]

$$\begin{array}{l}\mathsf{typing\_rename}: \\ \quad x \notin (\mathsf{dom}(E) \cup \mathsf{FV}(t_1)) \Rightarrow \quad E,\, x{:}T_1 \vdash_c t_1{}^x \;:\; T_2 \Rightarrow \\ \forall\, y \notin (\mathsf{dom}(E) \cup \mathsf{FV}(t_1)), \qquad E,\, y{:}T_1 \vdash_c t_1{}^y \;:\; T_2\end{array}$$

This renaming lemma allows us to go from a typing derivation for a single variable x, to a typing derivation for a cofinite set of variables. Using this lemma and choosing $L' = \mathsf{dom}(E) \cup \mathsf{FV}(t_1)$ lets us complete the decidability proof.

The renaming lemma is simply a consequence of properties (substitution and weakening) that we have already proven about the cofinite typing judgement. Importantly, showing these properties does not require renaming. We can derive typing_rename as follows. If $x = y$, then there is nothing to do. Otherwise, we rewrite the conclusion using subst_intro as

$$E,\, y{:}T_1 \vdash_c [x \to \mathsf{fvar}\ y]\,(t_1{}^x) \;:\; T_2\,.$$

By the substitution lemma, typing_subst, it suffices to show that

1. $E,\, y{:}T_1 \vdash_c \mathsf{fvar}\ y \;:\; T_1$ and
2. $E,\, y{:}T_1,\, x{:}T_1 \vdash_c t_1{}^x \;:\; T_2\,.$

To show (1), we use C-TYPING-VAR. To show (2), we use the weakening lemma, by which it suffices to show that $E,\, x{:}T_1 \vdash_c t^x \;:\; T_2$. This result follows from the assumptions of typing_rename.

In our experience, renaming lemmas are rarely needed. Aside from typing_decidable, the only situations in our developments where they are required are in the proofs of confluence for beta reduction in the lambda calculus and the Calculus of Constructions. In these proofs, the conlusion of the theorem is an existential statement: "there exists a common reduct…." The cases which require renaming involve rules with binding, where we have an induction hypothesis of the form "for all x not in L', there exists t such that …" which we need to use to prove a statement of the form "there exists t such that for all y not in L, ….". The induction hypothesis is strictly weaker than the statement we are trying to prove. For each name x, it gives us an object t for which some relation holds. We must use the renaming lemma for that relation to show that the relation holds for all sufficiently fresh names y.

4.3 Equivalence of Exists-Fresh and Cofinite Definitions

At this point, we have defined two sets of rules for $\lambda_{\to}$: one consisting of the original exists-fresh definitions, which corresponds directly to the "paper presentation," and one that uses cofinite quantification, which yields stronger induction principles. To be confident that the cofinite definitions are adequate, we verify (formally) that they define the same relations as the exists-fresh variants. [9] Note that these proofs of equivalence are straightforward consequences of renaming lemmas.

Let us take the typing relation as an example. The theorem to prove is the following:

$$E \vdash_c t \;:\; T \quad \Leftrightarrow \quad E \vdash t \;:\; T\,.$$

Both directions are straightforward inductions. The TYPING-ABS case of the $\Leftarrow$ direction is the only interesting case, where we must show $(E \vdash_c \mathsf{abs}\ t_1 \;:\; T_1 \to T_2)$ from the induction hypothesis $(E,\, x{:}T_1 \vdash_c t_1{}^x \;:\; T_2)$ and an assumption $(x \notin \mathsf{FV}(t_1))$. This case follows by first applying C-TYPING-ABS, instantiating L to be $(\mathsf{FV}(t_1) \cup \mathsf{dom}(E) \cup \{x\})$, and then using typing_rename.

4.4 Streamlining Developments

In summary, the infrastructure of our developments includes the following steps.

1. State definitions using exists-fresh quantification.
2. Restate the definitions using cofinite quantification.
3. Prove basic lemmas, such as subst_intro.
4. Prove substitution and weakening lemmas.
5. Prove renaming lemmas.
6. Prove the equivalence of the cofinite and exists-fresh presentations.

We can actually get by with less. To begin with, observe that the exists-fresh definitions are needed only in the first and last steps above. The main reason for stating the exists-fresh definitions is to check (informally) that the cofinite definitions adequately encode some set of informal mathematical definitions and statements of interest. Thus, it is only strictly necessary to state the exists-fresh definitions for relations whose adequacy is crucial—i.e., those relations whose definitions are necessary to state the main theorems of interest or that appear hereditarily in those theorems. The style can be streamlined by omitting steps 1 and 6 for definitions of technical relations that are introduced only to help with proofs.

In fact, the style can be streamlined further, according to need and personal taste. In particular, one may (after gaining some familiarity) feel convinced that the cofinite definitions are adequate encodings of one's informal ideas, without the need of a formal proof of equivalence with their "more obviously adequate" exists-fresh versions—and, just as importantly, one may be confident that one's *readers* will feel similarly.[10] In this case, steps 1 and 6 can be completely omitted.

Finally, if we omit step 6, we can then prove renaming lemmas—step 5—only as needed, rather than going to the trouble of stating and proving them for all relations. Indeed, as we described in Section 4.2, our experience has been that they are rarely needed.

In short, the core of our style is to define relations using cofinite quantification and to reason about those particular definitions, i.e., steps 2 through 4. The remaining steps need not be carried out for every relation.

4.5 Comparison

As we described in Section 2, we are not the first to propose a locally nameless representation for languages with binders. Nevertheless, our style of formal reasoning using this representation differs from previous work in subtle but important ways. As a result, our style requires much less infrastructure than alternative approaches.

In this subsection, we give a detailed comparison to the styles used by Gordon (1994), McKinna and Pollack (1993; 1999), and Leroy (2007). Note that, to obtain a meaningful comparison, we sometimes must apply their styles to our definitions and talk about rules not mentioned in their papers.

Gordon was the first to use a locally nameless representation for formal reasoning about binders. In contrast to our typing judgements (both the exists-fresh and cofinite versions), he stated his

[8] While the freshness condition on $y \notin \mathsf{dom}(E)$ ensures that $(E,\, y{:}T_1)$ is well-formed, the freshness condition $x \notin \mathsf{dom}(E)$ could be derived with some work from the second hypothesis; nevertheless we we choose to stick to a simpler and symmetric presentation.

[9] The formal proofs can be found in file `STLC_Core_Adequacy.v`.

[10] It is worth emphasizing that the exists-fresh versions *are* more obviously adequate, in a technical sense: they can be formalized in a very weak theory—Primitive Recursive Arithmetic—while cofinite presentations involve rules with "infinitely many hypotheses" that cannot be formalized in any conservative extension of PRA.

judgement using the close operation instead of open:

$$\frac{E,\, x{:}S \vdash t \;:\; T}{E \vdash \mathsf{abs}\ (^{\backslash x}t) \;:\; S \to T}$$

Gordon observed that the default induction principle arising from rules in this form it not very strong. In particular, it cannot directly prove the weakening lemma. As with the exists-fresh version, the induction hypothesis for the abstraction case only applies to a single variable, about which we know nothing.

Therefore, Gordon manually strengthened the induction principle for his language by introducing a finite set L of variables that x must be fresh for in the binding case. This L is an additional argument to the induction principle, rather than an existentially bound variable in the typing rule for abstractions. Gordon derived this principle from the default one by using induction on the height of the typing derivation. In developments, he uses this strengthened principle to eliminate typing judgments, while preserving the rule above as an introduction form.

Although there is a finite set involved, and it solves the problem of weakening in a similar manner, Gordon's strengthened induction principle is not exactly the same as the one arising from our form of cofinite quantification, since the induction hypothesis only applies to a single variable in the abstraction case. The change is that we get to choose a finite set of names for which that variable is assumed to be fresh. In this respect, Gordon's strengthened principle is more similar to that of Urban et al. (2007b) than to ours.

The important difference between our style and Gordon's is in terms of infrastructure. Gordon's style requires stating a strengthened induction principle for each relation involving binders and defining the height of that relation to prove the soundness of the induction principle. In contrast, our strong induction principle is automatically derived from the (single) cofinite definition of the relation (and therefore can automatically be used with Coq's induction tactics). Furthermore, our cofinite definitions also automatically provide strong inversion principles, whereas Gordon must state and prove these manually. If necessary, we may show our strengthened rules equivalent to that produced by the exists-fresh rule using standard results, weakening and substitution.

Also, Gordon's style may be more difficult to work with. His use of the close operation suffers from the same problem with equivalence as the nominal approach: $\mathsf{abs}\ (^{\backslash x}t) = \mathsf{abs}\ (^{\backslash y}u)$ does not imply that $x = y$ and $t = u$. Furthermore, to use Gordon's typing rule in a backwards proof one might have to coerce the goal into the correct form.

McKinna and Pollack, in their work on formalizing Pure Type Systems, also proposed a style of development for working with locally named terms. The most significant difference from our style is that it obtains strengthened induction principles by stating *universal* definitions of inductive relations, which quantify bound variables over as many names as possible. For example, they would state the typing rule for abstractions as follows:

$$\frac{\forall\, x \notin \mathsf{dom}(E),\ \ (E, x{:}S \vdash_a t^x \;:\; T)}{E \vdash_a \mathsf{abs}\ t \;:\; S \to T}$$

With this form of quantification, proving metatheoretic results—step 4 in our style—must, in general, come *after* the equivalence between exists-fresh and universal definitions has been proven. The substitution lemma, which we rely on to prove the equivalence between exists-fresh and cofinite definitions, cannot be proven directly for the universal definition of a relation. Thus, our strategy for proving equivalence between definitions, which piggybacks on standard metatheoretic results, fails in the McKinna-Pollack style. Instead, they must prove equivalence using swapping, at significant cost in infrastructure. In particular, the operation of swapping names must be defined for every entity (context, terms, etc.) and all relations must be shown to be equivariant (i.e. stable under swapping).

Since exists-fresh definitions alone cannot be used to prove weakening, and universal definitions alone cannot be used to prove substitution, the McKinna-Pollack style is, in general, to use both sets of definitions. The induction principle from universal definition provides a strong elimination form (e.g., that allows weakening to be proven) whereas the rules from exists-fresh definitions are strong introduction forms (e.g., that allow substitution to be proven).

Leroy used McKinna and Pollack's ideas in a locally nameless solution to the POPLMARK Challenge. As he demonstrates, it is possible to completely omit exists-fresh definitions from a development, thus streamlining the McKinna-Pollack style as we streamlined ours by dropping steps 1 and 6. Leroy takes universal definitions as the preferred ones, thus obtaining strong induction principles "for free," and proves that exists-fresh forms of rules are admissible for the universal relation. For example, he proves the following lemma for the typing relation:

$$\mathsf{t\_abs'}: \quad E, x{:}S \vdash_a t^x \;:\; T \quad \Rightarrow \quad x \notin \mathsf{FV}(t) \quad \Rightarrow \quad E \vdash_a \mathsf{abs}\ t \;:\; S \to T\,.$$

However, such lemmas must be stated for every rule with binding in the development, and the proof of such lemmas requires the significant amount of infrastructure related to swapping.

5. Practical Formal Metatheory

We close with a concrete discussion of our Coq realization of working with cofinite quantification. We present the major developments we have carried out—System $F_{<:}$, the Calculus of Constructions, and extensions of ML—to demonstrate that our techniques scale to larger languages and are expressive enough for wide use. We discuss issues specific to each development (e.g., dealing with multi-binders) as well as properties common to all of them (e.g., the conciseness and robustness on change of our scripts). Finally, we give an overview of the structure of our developments for readers who may wish to use them as a basis for their own formalizations.

5.1 System $F_{<:}$

The heart of the POPLMARK Challenge is a proof of type soundness of System $F_{<:}$. This task was designed as a stress test for formalized metatheory. Our solution closely follows the structure proposed in the appendix of the challenge's description.

To get a sense of how our solution compares to other solutions, we measured the number of lemmas and *steps of reasoning* involved in several developments;[11] the results are summarized in Figure 5. Our study is restricted to Coq developments since there are no meaningful metrics across different proof assistants. In this comparison, a step of reasoning is defined as the invocation of one tactic, excluding trivial ones: `intros` (introduces hypotheses into the context), `auto` (performs automated proof search), and simple variations of these two. We also exclude from the counts material from libraries that are reusable across metatheory developments. We isolate part 1A of the challenge (technical properties of the subtyping relation) from parts 1A+2A (preservation and progress) because some submissions cover only part 1A.

Even when attention is restricted to Coq developments, the comparison is necessarily rough, since the developments were carried out by different people with different preferences in the amount of automation they used, etc. Nevertheless, a few interesting points emerge. The first line of the table shows that our style compares well against Vouillon's development, which uses pure de Bruijn indices—a representation commonly thought to be quite effective.

[11] All are publicly available from `http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/`.

Author (chronological order)	Representation used	Lemmas 1A	Proof steps 1A	Lemmas 1A+2A	Proof steps 1A+2A
Vouillon	de Bruijn	30	402	72	1175
Leroy	locally nameless	49	495	128	1364
Stump	levels/names	56	938	-	-
Hirschowitz & Maggesi	de Bruijn (nested datatype)	49	1574	-	-
Chlipala	locally nameless	23	75	-	-
Our development	locally nameless	22	101	65	576

Figure 5. Comparison of Coq submissions to the POPLMARK Challenge.

Language	Results formalized	Number of trusted definitions	Infrastructure lemmas	Core lemmas	Infrastructure proof steps	Core proof steps
$\lambda_{\rightarrow}$	Preservation and progress	13	13	4	49	45
System $F_{<:}$	Preservation and progress	20	48	17	335	241
ML core only	Preservation and progress	23	37	8	181	98
ML with features	Preservation and progress	42	55	18	367	396
Lambda calculus	Church-Rosser	11	21	25	112	160
CoC	Church-Rosser, preservation	17	36	55	214	475

Figure 6. Complexity of our developments.

The solutions that do not use a standard de Bruijn or locally nameless representation (Stump and Hirschowitz & Maggesi) require significantly more steps. The last line shows that our style requires significantly less infrastructure than Leroy's solution, a locally nameless solution faithful to McKinna and Pollack's way of dealing with the quantification of names. Chlipala's is a variation on an earlier version of Leroy's development and is the only one that is shorter than ours, mainly because Chlipala's proof script relies heavily on development-specific tactics.

Although clever automation and development-specific tactics can make scripts extremely small (Nipkow [2001] gives an impressively automated proof of confluence, for example), they can also make them slow (hard to work with) and brittle (hard to reuse). Brittleness emerges when automation fails after some changes to earlier definitions or lemmas: it can be hard to know exactly what has been broken and how to fix it. Of course, the other extreme of no automation at all is also unmanageable, especially for metatheory proofs, which involve many related arguments. Moreover, too little automation can also make scripts brittle: even a tiny change in definitions or statements of lemmas will require update. Aiming for maximal robustness, our convention is to rely as much as possible on automation for low-level details (e.g., proving that a term is locally closed) while giving all the "important" arguments (the ones that would appear in a paper proof) manually.

5.2 Calculus of Constructions (CoC)

This formalization contains three main components. The first is a proof of confluence for parallel beta-reduction. Its use of the variable closing operation requires slightly more advanced reasoning about binders. We have also used this development to evaluate our scripts for robustness: we first completed a proof of confluence for the pure lambda calculus and then migrated it to CoC, preserving the same flow of arguments and just adding the extra cases and updating the names of some hypotheses.

The second part consists of a set of inversion results for the typing and the conversion judgments, which are quite technical in themselves but do not involve much difficulty with respect to binders. This part is representative of the effort required for the CoC formalization: we felt that the largest share of the work was to understand and be able to state clearly the arguments involved. The fact that the proofs were machine-checked appeared to be only a way to force oneself to understand all the details of the proofs formalized, not as an extra burden to be dealt with.

The third and last part is the core of the preservation proof, which shares a significant amount of structure with the corresponding proof for System $F_{<:}$ (only the steps of reasoning are more complex to follow). Compared to the System $F_{<:}$ development, the fact that terms and types are represented in a uniform way in CoC and that proofs are more complex reduces the binding infrastructure from about 60% of the total development down to 35% percent.

Compared to Barras's formalization of CoC in Coq (1997) using a pure de Bruijn representation, our style saves us from numerous issues associated with shifting indices—particularly bothersome when working with CoC's dependent types—and so, contrary to Barras, we did not need clever engineering of the statements of lemmas to make them fit the representation used.

5.3 Extensions of ML

We also investigated how to extend the basic $\lambda_{\rightarrow}$ development with features such as datatypes, fixpoints, references, exceptions, pattern matching, and ML-style polymorphic types. Then, building on the individual successes of each of these extensions, we set up the formalization of a language containing all of these features together. This experiment demonstrates that our style can be applied to full-featured programming languages, not just tiny lambda-calculi.

Pleasingly, most of the arguments that would be omitted as "trivial" on paper and which do not involve binders are solved by the `auto` tactic. In particular, the infrastructure lemmas do not need any change when we add features like pairs or references. The key proofs—preservation and progress—contain respectively in 60 and 80 lines, which seems reasonable for a system with 24 typing rules.

A key idea involved in this development is a treatment of *multi-binders*, used to encode ML type schemes, patterns, and recursion. The basic operation on a multi-binder is to open it with a *list* of terms. A bound variable pointing to the jth variable bound by the ith binder above the current position is represented as $(\mathsf{trm\_bvar}\, i\, j)$. Free variables are still represented as one single name and substitution lemmas are still stated in terms of an atomic substitution from one name to one term. Then, to prove subject reduction, for example, the substitution lemma is applied iteratively in the cases involving multi-binders.

5.4 Organization of our developments

Each of our developments is divided into three main parts: *trusted definitions*, *infrastructure*, and *core lemmas*.

The *trusted definitions* part contains the description of the language formalized—the syntax of the language, its semantics, and its type system, if any—and the statements of the main theorems which are to be proven about the language. This part is the only material that needs to be trusted, in the following sense: if one is convinced about the adequacy of these definitions and trusts that Coq correctly type-checks all proofs in the development, then one can have confidence that the system has been formally certified.

The *infrastructure* part sets up the machinery required for the *core lemmas* and consists of several components:

1. Language-specific specializations of tactics for working with cofinite quantification, e.g., to automatically choose a set L when applying a rule that uses cofinite quantification.
2. Proofs about properties of substitution (Figure 2).
3. Proofs that local closure is preserved by various operations, e.g., substitution (Section 3.3).
4. *Regularity lemmas* which state that relations contain only locally closed terms (Section 3.3).
5. Hints to enable Coq's automation to use regularity lemmas.

Note that the lemmas about substitution are similar from one language to the next. When setting up a new development, those who wish to get to "interesting" proofs quickly may state as axioms properties of substitution and regularity lemmas, proving them only after they believe that their main proofs will go through. In this manner, one can "test-drive" a language definition without having to revise infrastructure proofs as the definition is modified.

Finally, the *core lemmas* part contains the lemmas that would normally be stated in an informal presentation. The statements closely match their informal counterparts, and the proofs contain few uninteresting details—facts about local closure and freshness side conditions are almost always handled automatically.

We conclude this section with a breakdown of our developments by the number of definitions and lemmas in each part, as well as the number of proof steps required to prove the lemmas; the results are summarized in Figure 6. The number of trusted definitions gives an idea of the size of the language formalized. The number of core lemmas gives an idea of the theoretical complexity of the development; recall that these lemmas correspond closely with results that would be proven informally. The amount of infrastructure, in terms of both lemmas and steps, is proportional to the number of binding constructs and relations in the language. A comparison between figures from the last two columns suggests that the amount of infrastructure work is reasonable compared to the core proof work—especially since infrastructure proofs follow a standard pattern and are easily set up. More precisely, the ratio "infrastructure over core proofs" increases with the number of binding constructs involved but decreases with the inherent complexity of the development.

6. Conclusion

We have argued for a novel style for formalizing programming-language metatheory, based on a combination of of locally nameless representation with a cofinite idiom for quantifying free names in inductive definitions.

This design satisfies the evaluation criteria of POPLMARK. First, our presentation is transparent: the proofs closely follow their informal equivalents. Second the overheads of the approach are low: we do not need manual proofs of freshness side-conditions nor reasoning about alpha-equivalence, and only on rare occasions do we need to justify well-formedness of objects (e.g. local closure) or explicitly rename variables. When we do, the required proofs of renaming lemmas follow with virtually no infrastructure. At the same time, there is no need for external tools, and the style works in any general purpose theorem prover (although we found Coq to be well-suited to the task). In our scripts, definitions and results about variables, freshness, and environments are factored into a reusable library. Finally, experience with a number of large developments suggests that the approach is "complete" in the informal sense that any language definition and accompanying reasoning techniques that would be accepted as informally correct can be carried out in this style. Our formalizations provide a good starting point for new work—reports from early adopters confirm that modifying them is much easier than starting a new development from scratch.

In the future, we would like to integrate our approach with the Ott tool (Sewell et al. 2007), which automatically generates Coq (and LaTeX) definitions from compact, high-level language descriptions. A more ambitious next step is to study the possibility of automatically generating the required infrastructure proofs, which follow a fairly simple pattern. Finally, we would like to explore certified programming of tools such as type checkers and compilers that must deal with binders. In the long term, we hope that our techniques will help with the verification of both specifications and implementations of programming languages.

Acknowledgments Many thanks to the members of the Penn PLClub for extensive feedback and discussion, and to Michael Norrish and Christian Urban for important technical insights.

References

Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In Bezem and Groote (1993), pages 13–28.

Andrew W. Appel. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS), Boston, Massachusetts*, pages 247–258, June 2001.

Michael Ashley-Rollman, Karl Crary, and Robert Harper. Submission to the POPLMARK challenge, parts 1 and 2. Available from `http://www.cis.upenn.edu/~plclub/mmm/`, 2005.

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, *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

Henk P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

Bruno Barras and Benjamin Werner. Coq in coq. Available from `http://pauillac.inria.fr/~barras/coq_work-eng.html`, 1997.

M. Bezem and J. F. Groote, editors. *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications, TLCA '93*, volume 664 of *Lecture Notes in Computer Science*, 1993. Springer.

Anna Bucalo, Furio Honsell, Marino Miculan, Ivan Scagnetto, and Martin Hoffman. Consistency of the theory of contexts. *J. Funct. Program*, 16 (3), 2006.

Adam Chlipala. Submission to the POPLMARK challenge, part 1a. Available from `http://www.cs.berkeley.edu/~adamc/poplmark/`, 2006.

The Coq Development Team. The Coq proof assistant reference manual, version 8.1. Available from `http://coq.inria.fr/`, 2007.

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, 1991.

Karl Crary. Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212. ACM Press, 2003.

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.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995. Also available as INRIA Research Report 2556.

Jonathan M. Ford and Ian A. Mason. Operational techniques in PVS — A preliminary evaluation. *Electronic Notes in Theoretical Computer Science*, 42, 2001.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5): 341–363, July 2002.

Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J. J. Joyce and C.-J. H. Seger, editors, *Higher-order Logic Theorem Proving And Its Applications, Proceedings, 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 414–426. Springer, 1994.

Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 1996.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 2007. To appear. Available from `http://www.cs.cmu.edu/~drl/`.

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

Dimitri Hendriks and Vincent van Oostrom. Adbmal. In F. Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 136–150. Springer–Verlag, 2003.

Peter Homeier. A proof of the Church-Rosser theorem for the lambda calculus in higher order logic. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, pages 207–222. Division of Informatics, University of Edinburgh, September 2001. Available as Informatics Research Report EDI-INF-RR-0046.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. The theory of contexts for first order and higher order abstract syntax. *Electr. Notes Theor. Comput. Sci*, 62, 2001.

Gérard Huet. The constructive engine. In Raghavan Narasimhan, editor, *A Perspective in Theoretical Computer Science: Commerative Volume for Gift Siromoney*. World Scientific Publishing, 1989. Also available as INRIA Technical Report 110.

Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. Also available as INRIA Research Report 2009 (August 1993).

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

J. L. Krivine. *Lambda-Calculus, Types and Models*. Ellis Horwood, 1990.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184. ACM Press, 2007.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of the 33rd Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

Xavier Leroy. A locally nameless solution to the POPLmark challenge. Research report 6098, INRIA, January 2007.

Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

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. ACM Press, 2004.

James McKinna and Robert Pollack. Pure Type Systems formalized. In Bezem and Groote (1993), pages 289–305.

James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.

Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26(1):51–66, January 2001.

Michael Norrish and Konrad Slind. HOL 4. Available from `http://hol.sourceforge.net/`, 2007.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction, CADE 16: 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer, 1999.

Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

Randy Pollack. Reasoning about languages with binding: Can we do it yet?, February 2006. Presentation, slides available from `http://homepages.inf.ed.ac.uk/rpollack/`.

Robert Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *TYPES'93: Workshop on Types for Proofs and Programs, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1994a.

Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994b.

Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, March 1995.

Wilmer Ricciotti. Submission to the POPLMARK challenge, part 1a. Available from `http://ricciott.web.cs.unibo.it/`, 2007.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *ICFP '07: Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming*, 2007. To appear.

Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, 1988.

Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59 (3):317–325, 1988.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 2007. To appear; available from `http://www4.in.tum.de/~urbanc/publications.html`.

Christian Urban, Stefan Berghofer, and Julien Narboux. Nominal datatype package for Isabelle/HOL. Available from `http://isabelle.in.tum.de/nominal/`, 2007a.

Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21th Conference on Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007b.

René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. *Information and Computation*, 183(2):212–244, 2003.