The Calculus of Nominal Inductive Constructions

An Intensional Approach to Encoding Name-Bindings

Edwin Westbrook[1]
Rice University
emw4@rice.edu

Aaron Stump[2]
The University of Iowa
astump@cs.uiowa.edu

Evan Austin
The University of Kansas
ecaustin@ittc.ku.edu

Abstract

Although name-bindings are ubiquitous in computer science, they are well-known to be cumbersome to encode and reason about in logic and type theory. There are many proposed solutions to this problem in the literature, but most of these proposals, however, have been *extensional*, meaning they are defined in terms of other concepts in the theory. This makes it difficult to apply these proposals in intensional theories like the Calculus of Inductive Constructions, or CIC.

In this paper, we introduce an approach to encoding name-bindings that is *intensional*, as it attempts to capture the meaning of a name-binding in itself. This approach combines in a straightforward manner with CIC to form the Calculus of Nominal Inductive Constructions, or CNIC. CNIC supports induction over data containing bindings, comparing of names for equality, and associating meta-language types with names in a fashion similar to HOAS, features which have been shown difficult to support in practice.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Mechanical verification; F.4.1 [*Mathematical Logic*]: Mechanical theorem proving

1. Introduction

Name-bindings are ubiquitous in computer science. These are constructs, such as the λ-abstraction $\lambda x.\, t$ of the λ-calculus, that introduce a fresh name in a local scope. Unfortunately, name-bindings are well-known to be cumbersome to encode and reason about in logic and type theory, making it tedious to prove properties about programming languages and other formalisms that use name-bindings. This has led to much research into techniques for encoding name-bindings that mitigate this burden [16, 10, 8, 20, 7, 19, 13, 24, 21, 18, 4].

Most of these proposals define name-binding *extensionally*, meaning that name-binding is defined in terms of other concepts in the theory. Stated differently, name-binding is defined by giving a model for it in terms of other constructs, rather than by trying to define it directly. Nominal Logic, for example, defines name-bindings as α-equivalence classes, while Higher-Order Abstract Syntax (HOAS) defines name-bindings in terms of parametric functions. Extensional definitions work well in extensional type theories, and indeed, Nominal Logic has been used to much success in the Isabelle proof assistant [14] which is based on the extensional type theory HOL. None of these approaches, however, have been shown fully compatible with intensional theories such as the Calculus of Inductive Constructions (CIC), the basis of the Coq proof assistant [23]. Specifically, it is difficult in CIC to express recursion principles for data containing name-bindings under many of these approaches, and many of the approaches also require additional axioms to be added to CIC. In short, current extensional definitions do not quite capture the intended *meaning* of name-binding in intensional type theory.

In this paper, we introduce an approach to encoding name-bindings that is *intensional*, as it attempts to capture the meaning of a name-binding in itself. This is done by formalizing the meaning of a name-binding directly in the theory as a construct called a ν-abstraction.[1] We then show how the ν-abstraction can be combined with CIC to form a new theory which we call the Calculus of Nominal Inductive Constructions, or CNIC. CNIC supports all of CIC, and supports the following features related to name-binding: induction over data containing bindings; comparing of names for equality; and associating meta-language types with names in a fashion similar to HOAS. These features have been shown to be difficult to support for many previous approahces.

The remainder of this document is organized as follows. Section 2 introduces CNIC and its approach to name-bindings. Section 3 formalizes CNIC and states it meta-theoretic properties; proofs of these are given for a similar system in the first author's dissertation [26]. Section 4 discusses the current implementation of CNIC, called `cinic`, which is available from the first author's website, `http://www.cs.rice.edu/~emw4`. Section 5 describes a modest-sized example, a proof of confluence of the untyped λ-calculus, that has been formalized and checked in `cinic`. Section 6 then compares CNIC to related work and Section 7 concludes and discusses potential future work.

2. Name-Bindings in CNIC

In this section, we introduce the notion of name-bindings used here, show how it is modeled in CNIC, and then introduce the key features of CNIC for manipulating name-bindings. The notion of name-binding captured in CNIC is that of a construct, such as the λ-abstraction $\lambda x.\, t$ of the untyped λ-calculus, that introduces a fresh

[1] This author was partially supported by NSF grant CSR/EHS #0720857.

[2] This author was partially supported by NSF grant CCF #0841554.

[1] Using ν for name-bindings is mnemonic for a "new" name and seems to have originated with Odersky [15], though the current use is closer to that of Schürmann and Poswolsky [21].

name in a local scope. Specifically, the name-binding is defined by the following four properties:

- **Freshness**: The name that is introduced is distinct from any names bound outside the given binding, so that, for example, $x \neq y$ holds inside t in $\lambda x.\, \lambda y.\, t$;
- **α-equivalence**: Terms with name-bindings are equal up to renaming of bound names, so that $\lambda x.\, x$ equals $\lambda y.\, y$;
- **Scoping**: A name cannot occur outside a binding for it, so x is only a valid λ-term inside some binding for x; and
- **Typing**: Different types of names can be bound, such as the term and type variables of System F, or, in a typed representation of the simply-typed λ-calculus, $\lambda\, x : A\, .\, t$ binds x specifically as a variable of type A.

Note that scoping does not mean that CNIC cannot manipulate open terms; we allow the possibility that a name is bound outside of a λ-term, in the meta-language itself, to intermediate values in a computation that are open. In the end, though, we are only really interested in the closed λ-terms. Thus scoping is about *adequacy*, that is, that an encoding captures all and only the closed λ-terms. It is possible in CNIC to encode a λ-calculus over a given set of free variables, but that would be a different theory.

To support an intensional view of name-binding, CNIC introduces a construct called the ν-abstraction. Philosophically, ν-abstractions capture the intensional meaning of name-binding. The syntax of a ν-abstraction is $\nu\, \alpha : A\, .\, M$, which binds the name α of type $\mathsf{Name}\ A$ inside the term M. This construct supports the four name-binding properties by definition in CNIC: the name α is always guaranteed to be distinct from all other names in scope, so freshness holds; ν-abstractions are considered equal up to renaming of bound names, so α-equivalence holds; names cannot occur outside of bindings for them, so scoping holds; and names are introduced with some given type, so typing holds.

The user can make use of these properties by encoding name-bindings as ν-abstractions. Bound names are then encoded as CNIC names. For example, the following constructors define an encoding of the untyped λ-calculus as a type trm in CNIC, where the type $\nabla\alpha : A\, .\, B$ (note that ∇ is pronounced "nabla") is the type of a ν-abstraction $\nu\, \alpha{:}A\, .\, M$, where M has type B:

$$\begin{array}{lcl} \mathsf{var} & : & (\mathsf{Name\ trm}) \Rightarrow \mathsf{trm} \\ \mathsf{app} & : & \mathsf{trm} \Rightarrow \mathsf{trm} \Rightarrow \mathsf{trm} \\ \mathsf{lam} & : & (\nabla\alpha{:}\mathsf{trm}\, .\, \mathsf{trm}) \Rightarrow \mathsf{trm} \end{array}$$

These three constructors match the three cases in the definition of λ-terms. Bound variables x are encoded as $\mathsf{var}\ \alpha$, where α is a bound name of type $\mathsf{Name\ trm}$. Note the use of the typing property here: var can only be used with names specifically introduced to model λ-variables. Applications $t\ u$ are encoded as $\mathsf{app}\ M\ N$, where M and N are the encodings of t and u, respectively. The λ-abstraction $\lambda x.\, t$ is encoded as $\mathsf{lam}\ (\nu\, \alpha : \mathsf{trm}\, .\, M)$, where M is an encoding of t that uses $\mathsf{var}\ \alpha$ for x. Because λ-abstractions are encoded with ν-abstractions, they satisfy the four name-binding properties given above "for free." For example, α-equivalence of ν-abstractions ensures that any encodings of $\lambda x.\, x$ and $\lambda y.\, y$ are equal terms in CNIC.

The simplest tool for manipulating bindings in CNIC is called the *name replacement*. This construct allows the user to access the body of a ν-abstraction by passing in a fresh name to be used in place of the name bound by the ν-abstraction. Intuitively, this "peels off" the ν of a ν-abstraction. Syntactically, name replacements are written $M\ @\ \alpha$, and have the computation rule that $(\nu\, \beta : A\, .\, M)\ @\ \alpha$ evaluates to $[\alpha/\beta]M$. Note that this is not safe if α is used in M to begin with, since β needs to be distinct from α in M by freshness. Viewed differently, name replacements treat ν-abstractions as partial functions, whose domain is the set of names that are fresh. This is similar to the concretion operator of [4].

Names can be compared in CNIC with *name-matching functions*. These are similar to pattern-matching functions, but operate on terms of type $\mathsf{Name}\ A$. A name-matching function must have a case for every name in scope, since an unknown value of type $\mathsf{Name}\ A$ could potentially be equal to any of the names in scope. Name-matching functions also must always have a catch-all case that matches any name not in scope, since evaluation could move the name-matching function into a larger scope with more names than the one in which it was written. As an example, the CNIC term

$$\begin{array}{l} \mathsf{nfun}\ (\alpha \setminus\ \rightarrow \mathsf{true} \\ \qquad\quad \mid \beta \setminus U : \mathsf{Type}_0, \beta : U \rightarrow \mathsf{false}) \end{array}$$

is a function that takes in a name and compares the name with α, which must be the only name currently in scope. The notation $M \setminus \Gamma$ is a pattern that matches any term that is a substitution instance of M for the variables and names listed in Γ. Thus the first case matches the name α and returns true, while the second case matches any other name β of type $\mathsf{Name}\ U$ for some U and returns false.

CNIC also contains a specialized form of name-matching function over names of type $\mathsf{Name}\ A$ where A is inductive. This form eliminates cases for names of different types, and also refines the type of the catch-all case. For example, the CNIC term

$$\begin{array}{l} \mathsf{nfun}\ (\alpha \setminus\ \rightarrow \mathsf{true} \\ \qquad\quad \mid \beta \setminus \beta : \mathsf{trm} \rightarrow \mathsf{false}) \end{array}$$

matches over names of type $\mathsf{Name\ trm}$, where α is the only name in scope with type $\mathsf{Name\ trm}$. There could be names besides α in scope with types other than $\mathsf{Name\ trm}$. CNIC, however, requires that all these other names have type $\mathsf{Name}\ B$ for some B that is also inductive, in order to simplify the type refinement for the catch-all case.

A final feature of CNIC is that it allows pattern-matching inside ν-abstractions. For example, if we define the type nat of natural numbers with the usual constructors zero and succ, then a value of type nat should not have any names free. The following function witnesses this fact by lifting a natural number out of a ν-abstraction:

$$\begin{array}{l} \mathsf{fun\ lift\text{-}nat}\ (\nu\, \alpha{:}A\, .\, \mathsf{zero} \setminus\ \rightarrow \mathsf{zero} \\ \qquad\qquad\quad \mid \nu\, \alpha{:}A\, .\, \mathsf{succ}\ (x\ @\ \alpha) \setminus x : \nabla\alpha{:}A\, .\, \mathsf{nat} \rightarrow \\ \qquad\qquad\qquad \mathsf{succ}\ (\mathsf{lift\text{-}nat}\ x)) \end{array}$$

For example, lift-nat applied to $\nu\, x : A\, .\, \mathsf{succ\ \ zero}$ returns $\mathsf{succ\ \ zero}$. Note that the pattern variable x in the case for succ is applied to α with a name replacement. This is because α is bound in the pattern and x is bound outside it, and so any value substituted for x cannot contain α. Stated differently, the pattern $\nu\, \alpha : A\, .\, \mathsf{succ}\ x$ only matches terms $\nu\, \alpha : A\, .\, \mathsf{succ}\ M$ where M does not contain α, excluding the possibility that M is, e.g., $y\ \alpha$ for some free variable y. Since this pattern is less general than $\nu\, \alpha : A\, .\, \mathsf{succ}\ (x\ @\ \alpha)$, and because allowing the former would require a complex occurs check at runtime, CNIC simply requires all pattern variables to be fully applied using name replacements to the names bound in the pattern.

3. CNIC Formalized

This section formalizes the syntax and semantics of CNIC and states some of its meta-theoretic properties. The syntax of CNIC is given in Figure 1. Many of the term constructs come directly from CIC, including the predicative universes Type_i (where we here use Type_0 for the universe Set), the abstraction type $\Pi x : A\, .\, B$, inductive type constructors a, variables x, constructors c, λ-abstractions $\lambda\, x : A\, .\, M$, and applications $M_1\ \ M_2$. Most of

$$
\begin{array}{rcl}
M & ::= & \mathsf{Type}_i \mid \mathsf{Prop} \mid \Pi x\!:\!A\,.\,B \mid \nabla\alpha\!:\!A\,.\,B \mid a \mid \mathsf{Name} \\
 & & \mid u \mid x \mid c \mid \alpha \mid \nu\,\alpha\!:\!A\,.\,M \mid M \mathbin{@} \alpha \mid \lambda\,x\!:\!A\,.\,M \\
 & & \mid M\ M \mid \mathsf{nfun}\ (\Gamma)\ (P^{\alpha} \rightarrow M \mid \ldots \mid P^{\alpha} \rightarrow M) \\
 & & \mid \mathsf{fun}\ u\ (\Gamma)\ (P^{c} \rightarrow M \mid \ldots \mid P^{c} \rightarrow M) \\
P^{c} & ::= & \nu\,\Gamma^{\alpha}\,.\,c\,x \mathbin{@} \Gamma^{\alpha} \ldots x \mathbin{@} \Gamma^{\alpha} \setminus \Gamma \\
P^{\alpha} & ::= & \nu\,\Gamma^{\alpha}\,.\,\alpha \setminus \Gamma \\
\Gamma & ::= & \Gamma, x : A \mid \Gamma, \alpha : A \mid \cdot \\
\Sigma & ::= & \Sigma, c : A \mid \Sigma, a : A \mid \Sigma, u : A \mid \cdot \\
\sigma & ::= & [M/x, \sigma] \mid \cdot
\end{array}
$$

Figure 1. Syntax of CNIC

the new term constructs have been introduced above, including nabla-types, names, ν-abstractions, name replacments, and name-matching functions. The pattern-matches and recursive functions of CIC are combined into the single pattern-matching function construct, which has also been introduced above. Note that the general form of name- and pattern-matching functions given here contain an extra context Γ before the patterns of the function, which specifies an additional list of parameters to the function.

Note that patterns for pattern- and name-matching functions can contain ν-abstractions for an arbitrary list of names. This is written as $\nu\,\Gamma^{\alpha}\,.\,M$, where the notation Γ^{α} is used to denote a context containing only names. The variables occurring in patterns for pattern-matching functions must be applied to these names using name replacements, which is denoted by the notation $x \mathbin{@} \Gamma^{\alpha}$.

CNIC contexts Γ differ from CIC contexts in that contexts in CNIC assign types both to variables and names. CNIC also includes modal variables u and modal contexts Σ. These are included for purely technical reasons: name- and pattern-matching functions are always considered closed, that is, all of the variables and names they use must be bound either in their patterns or in their list of parameters. The exception is that they are allowed to contain recursive calls to enclosing pattern-matching functions. Thus the variables used for recursive calls to pattern-matching functions are always modal variables u, and we see below that the typing rules for pattern- and name-matching functions then discard all other free variables and names in the normal context Γ. Modal contexts are also used to ascribe types to constructors and type constructors.

The remainder of this section uses the following notations, some of which were briefly introduced above. We use Γ^{x} and Γ^{α} to denote contexts that only bind variables or names, respectively. We use Γ_{p} for normal CNIC contexts that are used as parameter lists in pattern- and name-matching functions. We use R for an argument, which is either a term M or the syntax $@\ \alpha$ for some α. We write $N\ R$ to denote either the application to or name replacement of R, depending on the form of R. We use $\vec{R}$ to denote a list of arguments, where R_i denotes the ith element of this list and $M\ \vec{R}$ denotes the application to and/or name replacement of each argument R_i in order. We use $M\ \Gamma$ to denote the application and/or name replacement of M to each variable or name bound in Γ. We use $\Pi\Gamma\,.\,A$ and $\nabla\Gamma^{\alpha}\,.\,A$ to denote Π- and ∇-abstractions of the variables and names in Γ or Γ^{α}, respectively. We use $\nu\,\Gamma^{\alpha}\,.\,M$ for ν-abstractions of the names in Γ^{α}. We use $[\vec{N}/\vec{x}]$ to denote the substitution $[N_1/x_1, \ldots, N_n/x_n]$, and we use $[\vec{R}/\Gamma]$ to denote combined substitution for variables in Γ by terms in $\vec{R}$ and renaming of names in Γ with names in $\vec{R}$.

The notation $(M)\!\uparrow^{\Gamma^{\alpha}}$ denotes the raising of M by Γ^{α}, which yields the term $\nu\,\Gamma^{\alpha}\,.\,[(x_1 \mathbin{@} \Gamma^{\alpha})/x_1, \ldots, (x_n \mathbin{@} \Gamma^{\alpha})/x_n]M$ for all free variables x_i in M. Similarly, $(M)\!\uparrow^{\Gamma^{\alpha}}_{\nabla}$ denotes the raising of M by Γ^{α} as a type, which yields a similar term with $\nu\,\Gamma\,.$ replaced by $\nabla\Gamma\,.$. The notation $(\Gamma)\uparrow^{\Gamma^{\alpha}}$ raises each type in Γ by Γ^{α} as a type.

The operational semantics of CNIC is given in Figure 2. This is given as a list of rewrite rules. These are mostly straightforward. The first rule applies one step of a pattern-matching function on a constructor application. The second and third rules apply name-matching functions to either names bound in the function or unknown names that match the catch-all case. The fourth rule is standard β-reduction, while the fifth rule is a version of β-reduction for ν-abstrations. Note that name replacements $M \mathbin{@} \alpha$ are syntactically restricted so that α does not occur free in M, and thus the fifth rule always applies by α-equivalence when the body of a name replacement is a ν-abstraction. These rules define a Higher-Order Name-Binding Rewrite System that is orthogonal, and thus confluent; see the first author's dissertation [26].

Typing in CNIC is given by the judgment $\Sigma; \Gamma \vdash M : A$. This judgment implicitly assumes well-formedness of the modal context Σ and well-formedness relative to Σ of the context Γ. These judgments are straightforward; the latter requires only that the types in Γ are well-typed, while the former requires both this condition and that the given inductive types satisfy the positivity and universe constraints of CIC [23].

The rules for the typing judgment of CNIC are given in Figure 3. These use s to denote a sort, that is, a term that is either Prop or Type_i for some i. Many of these rules are standard in CIC, and so we do not discuss them here. Also standard is the subtyping judgment $\vdash A \lesssim B$, which is used in the conversion typing rule to capture universe inclusion. The CNIC definition of subtyping is a straightforward extension of subtyping in CIC to include a case for ∇-types, so we omit it here. Examining the rules that are new in CNIC, the predicative and impredicative typing rules for ∇-types mirror those for Π-types. The rule for $\mathsf{Name}\ A$ requires A to be well-typed. The rule for names α looks up α in the current context, adding Name to the returned type. The rule for ν-abstractions adds the bound name to the context and checks the body.

The rule for name replacements $M \mathbin{@} \alpha$ uses the operation $\mathbf{remove}_{\alpha}(\Gamma)$ to remove the name α from Γ when typing M. This is in order to ensure that α is fresh for M, as required. The operation $\mathbf{remove}_{\alpha}(\Gamma)$ also removes every name in Γ whose type contains α, so that Γ remains well-formed. In addition, all variables bound after α in Γ are also removed, since they could eventually have some term substituted for them that contains α, and thus a term containing such a variable is not guaranteed to be fresh for α.

The rule for pattern-matching functions is similar to what is found in CIC except for the presence of raisings, which are necessary to match under ν-abstractions. The type of the scrutinee is in general of the form $\nabla\Gamma^{\alpha}\,.\,a\ M_1 \ldots M_n$ for some M_i that could contain names in Γ^{α}. To abstract out the M_i, we match over the type $\nabla\Gamma^{\alpha}\,.\,a\ (M_1 \mathbin{@} \Gamma^{\alpha}) \ldots (M_n \mathbin{@} \Gamma^{\alpha})$, more succinctly written as $(a\ \Gamma^{\mathrm{x}})\!\uparrow^{\Gamma^{\alpha}}_{\nabla}$ for some context Γ^{x}. For this to be well-typed, the types of the variables in Γ^{x} must also be raised, so our function has type $\Pi\Gamma_{\mathrm{p}}\,.\,\Pi(\Gamma^{\mathrm{x}})\!\uparrow^{\Gamma^{\alpha}}\,.\,\Pi x\!:\!((a\ \Gamma^{\mathrm{x}})\!\uparrow^{\Gamma^{\alpha}}_{\nabla})\,.\,B$. The typing rule first checks that Γ^{x} is the argument context for a, then it checks that the return type is well-typed, and then it looks up the type $\Pi\Gamma^{\mathrm{x}}_{i}\,.\,a\ \vec{M}_i$ of each constructor c_i. Next it type-checks each body N_i with the modal variable u bound with the type of the whole function and with context $\Gamma_{\mathrm{p}}, (\Gamma^{\mathrm{x}}_{i})\uparrow^{\Gamma^{\alpha}}$, which includes the raised version of the argument context of c_i. The type computed must substitute the raised pattern and the raised type indices it induces for a into the expected return type B. Finally, the function must have a case for every constructor of a, written $\Gamma \vdash \vec{c}\ \mathbf{covers}\ a$, each case must pass termination checking, written $\vdash \mathbf{app\text{-}check}_{u}(\Gamma^{\mathrm{x}}_{i}; N_i)$, and the return type B must contain occurrences of x or variables in Γ^{x} that are fully applied with name replacements to at least as many names as are in Γ^{α}. Omitting this condition turns out to be equivalent to extensionality of ν-abstractions, which we do not necessarily wish to assume in CNIC at this time. The rules for name-

$$\begin{array}{rcl}
(\mathsf{fun}\ u\ (\Gamma_\mathrm{p})\ (\ldots \mid (c\ \Gamma){\uparrow}^{\Gamma^\alpha}\ \setminus \Gamma \to M_i \mid \ldots))\ \vec{R}\ (\nu\,\Gamma^\alpha\,.\,c_i\ \vec{N}) & \rightsquigarrow & [(\mathsf{fun}\ u\ (\Gamma_\mathrm{p})\ (\ldots))/u, (\nu\,\Gamma^\alpha\,.\,\vec{N})/\Gamma, \vec{R}/\Gamma_\mathrm{p}]M_i \\
(\mathsf{nfun}\ (\Gamma_\mathrm{p})\ (\ldots \mid \nu\,\Gamma^\alpha\,.\,\alpha_i \setminus \cdot \to \vec{M}_i \mid \ldots))\ \vec{R}\ (\nu\,\Gamma^\alpha\,.\,\alpha_i) & \rightsquigarrow & [\vec{R}/\Gamma_\mathrm{p}]M_i \\
(\mathsf{nfun}\ (\Gamma_\mathrm{p})\ (\ldots \mid \nu\,\Gamma^\alpha\,.\,\beta \setminus \beta : A \to \vec{M}_0 \mid \ldots))\ \vec{R}\ (\nu\,\Gamma^\alpha\,.\,\beta) & \rightsquigarrow & [\vec{R}/\Gamma_\mathrm{p}]M_0 \\
(\lambda\,x\!:\!A\,.\,M)\ N & \rightsquigarrow & [N/x]M \\
(\nu\,\alpha\!:\!A\,.\,M)\ @\ \alpha & \rightsquigarrow & M
\end{array}$$

Figure 2. Operational Semantics of CNIC

$$\dfrac{\Sigma;\Gamma \vdash M : A \quad \Sigma;\Gamma \vdash B : \mathsf{Type}_i \quad \vdash A \lesssim B}{\Sigma;\Gamma \vdash M : B} \qquad \dfrac{}{\Sigma;\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}} \qquad \dfrac{}{\Sigma;\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_1}$$

$$\dfrac{\Sigma;\Gamma \vdash A : \mathsf{Type}_i \quad \Sigma;\Gamma, x : A \vdash B : \mathsf{Type}_i}{\Sigma;\Gamma \vdash \Pi x\!:\!A\,.\,B : \mathsf{Type}_i} \qquad \dfrac{\Sigma;\Gamma \vdash A : s \quad \Sigma;\Gamma, x : A \vdash B : \mathsf{Prop}}{\Sigma;\Gamma \vdash \Pi x\!:\!A\,.\,B : \mathsf{Prop}}$$

$$\dfrac{\Sigma;\Gamma \vdash A : \mathsf{Type}_i \quad \Sigma;\Gamma, \alpha : A \vdash B : \mathsf{Type}_i}{\Sigma;\Gamma \vdash \nabla\alpha\!:\!A\,.\,B : \mathsf{Type}_i} \qquad \dfrac{\Sigma;\Gamma \vdash A : s \quad \Sigma;\Gamma, \alpha : A \vdash B : \mathsf{Prop}}{\Sigma;\Gamma \vdash \nabla\alpha\!:\!A\,.\,B : \mathsf{Prop}} \qquad \dfrac{a : A \in \Sigma}{\Sigma;\Gamma \vdash a : A} \qquad \dfrac{\Sigma;\Gamma \vdash A : s}{\Sigma;\Gamma \vdash \mathsf{Name}\ A : s}$$

$$\dfrac{x : A \in \Gamma}{\Sigma;\Gamma \vdash x : A} \qquad \dfrac{u : A \in \Sigma}{\Sigma;\Gamma \vdash u : A} \qquad \dfrac{c : A \in \Sigma}{\Sigma;\Gamma \vdash c : A} \qquad \dfrac{\alpha : A \in \Gamma}{\Sigma;\Gamma \vdash \alpha : \mathsf{Name}\ A} \qquad \dfrac{\Sigma;\Gamma, \alpha : A \vdash M : B}{\Sigma;\Gamma \vdash \nu\,\alpha\!:\!A\,.\,M : \nabla\alpha\!:\!A\,.\,B}$$

$$\dfrac{\Sigma;\Gamma \vdash \alpha : \mathsf{Name}\ A \quad \Sigma;\mathbf{remove}_\alpha(\Gamma) \vdash M : \nabla\alpha\!:\!A\,.\,B}{\Sigma;\Gamma \vdash M\ @\ \alpha : B} \qquad \dfrac{\Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda\,x\!:\!A\,.\,M : \Pi x\!:\!A\,.\,B} \qquad \dfrac{\Sigma;\Gamma \vdash M : \Pi x\!:\!A\,.\,B \quad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M\ N : [N/x]B}$$

$$\dfrac{\begin{array}{c} a : \Pi\Gamma^\mathrm{x}\,.\,s_1 \in \Sigma \qquad \Sigma;\cdot \vdash \Pi\Gamma_\mathrm{p}\,.\,\Pi(\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha}\,.\,\Pi x\!:\!(a\ \Gamma^\mathrm{x}){\uparrow}_{\nabla}^{\Gamma^\alpha}\,.\,B : s_2 \qquad \forall i(c_i : \Pi\Gamma^\mathrm{x}_i\,.\,a\ \vec{M}_i \in \Sigma) \\ \forall i(\Sigma, u : (\Pi\Gamma_\mathrm{p}\,.\,\Pi(\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha}\,.\,\Pi x\!:\!(a\ \Gamma^\mathrm{x}){\uparrow}_{\nabla}^{\Gamma^\alpha}\,.\,B)\ ;\ \Gamma_\mathrm{p}, (\Gamma^\mathrm{x}_i){\uparrow}^{\Gamma^\alpha} \vdash N_i : [(\vec{M}_i){\uparrow}^{\Gamma^\alpha}/\Gamma^\mathrm{x}\ ,\ (c_i\ \Gamma^\mathrm{x}_i){\uparrow}^{\Gamma^\alpha}/x]B) \\ \Gamma^\mathrm{x}, x \text{ fully applied w.r.t. } \Gamma^\alpha \text{ in } B \qquad \forall i(\vdash \mathbf{app\text{-}check}_u(\Gamma^\mathrm{x}_i; N_i)) \qquad \Gamma \vdash \vec{c}\ \mathbf{covers}\ a \end{array}}{\Sigma;\Gamma \vdash \mathsf{fun}\ u\ (\Gamma_\mathrm{p}, (\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha})\ (\ldots \mid (c_i\ \Gamma^\mathrm{x}_i){\uparrow}^{\Gamma^\alpha}\ \setminus (\vec{\Gamma^\mathrm{x}_c}){\uparrow}^{\Gamma^\alpha} \to \vec{N} \mid \ldots) : \Pi\Gamma_\mathrm{p}\,.\,\Pi(\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha}\,.\,\Pi x\!:\!((a\ \Gamma^\mathrm{x}){\uparrow}_{\nabla}^{\Gamma^\alpha})\,.\,B}$$

$$\dfrac{\begin{array}{c} \Sigma;\cdot \vdash \Pi\Gamma_\mathrm{p}, (\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha}\,.\,\Pi x\!:\!(\mathsf{Name}\ (a\ \Gamma^\mathrm{x})){\uparrow}_{\nabla}^{\Gamma^\alpha}\,.\,B : s \\ \forall i(\Sigma;\Gamma_\mathrm{p}, \Gamma^\alpha \vdash \alpha_i : \mathsf{Name}\ (a\ \vec{N})) \qquad \forall i(\Sigma;\Gamma_\mathrm{p} \vdash M_i : [\nu\,\Gamma^\alpha\,.\,\vec{N}/\Gamma^\mathrm{x}, \nu\,\Gamma^\alpha\,.\,\alpha_i/x]B) \\ \forall i(\Sigma;\Gamma_\mathrm{p}, \Gamma^\mathrm{x}, \beta : \mathsf{Name}\ (a\ \Gamma^\mathrm{x}) \vdash M_0 : [\nu\,\Gamma^\alpha\,.\,\Gamma^\mathrm{x}/\Gamma^\mathrm{x}, \nu\,\Gamma^\alpha\,.\,\beta/x]B) \qquad \Gamma^\mathrm{x}, x \text{ fully applied w.r.t. } \Gamma^\alpha \text{ in } B \end{array}}{\Sigma;\Gamma \vdash \begin{array}{l} \mathsf{nfun}\ (\Gamma_\mathrm{p}, (\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha})(\nu\,\Gamma^\alpha\,.\,\vec{\alpha} \setminus \cdot \to \vec{M} \\ \qquad \mid \nu\,\Gamma^\alpha\,.\,\beta \setminus \Gamma^\mathrm{x}, \beta : \mathsf{Name}\ (a\ \Gamma^\mathrm{x}) \to M_0) \end{array} : \Pi\Gamma_\mathrm{p}, (\Gamma^\mathrm{x}){\uparrow}^{\Gamma^\alpha}\,.\,\Pi x\!:\!((\mathsf{Name}\ (a\ \Gamma^\mathrm{x})){\uparrow}_{\nabla}^{\Gamma^\alpha})\,.\,B}$$

Figure 3. Typing for CNIC

matching functions are similar; note that we only include a rule for name-matching over Name of an inductive type.

We turn now to some meta-theoretic properties of CNIC. The following results have been proved in the first author's dissertation for a previous version of CNIC where all names had a single type Name, that is, where the Name type was not indexed by another type [26]. Adapting them to the current system is straightforward.

LEMMA 1 (Canonical Forms). *If $\Sigma;\cdot \vdash M : a\ \vec{N}$ for some Σ with no modal variables and some M in normal form, then M is of the form $c\ \vec{M}'$ for some constructor c of a.*

LEMMA 2 (Weakening). *If $\Sigma;\Gamma \vdash M : A$ then $\Sigma;\Gamma' \vdash M : A$ for any context Γ' containing all the variables and names of Γ and satisfying the property that if x comes before α in Γ then it also does in Γ'.*

LEMMA 3 (Substitution). *If $\Sigma;\Gamma_1 \vdash M : A$ and $\Sigma;\Gamma_1, x : A, \Gamma_2 \vdash N : B$ then $\Sigma;\Gamma_1, [M/x \vdash \Gamma :_2][M/x]N[M/x]B$.*

LEMMA 4 (Modal Substitution). *If $\Sigma;\Gamma_1 \vdash M : A$ and $\Sigma;\Gamma_1, u : A, \Gamma_2 \vdash N : B$ then $\Sigma;\Gamma_1, [M/u \vdash \Gamma :_2][M/u]N[M/u]B$.*

LEMMA 5 (Preservation). *If $\Gamma \vdash M : A$ and $M \rightsquigarrow N$, then $\Gamma \vdash N : A$.*

Consistency and strong normalization were also proved in the first author's dissertation for the system with a single type Name of names. This was done with a translation from the old version of CNIC into CIC that was shown to preserve reductions. We conjecture that this approach can be adapted in a straightforward manner to the current system.

CONJECTURE 1 (Strong Normalization). *The relation $\rightsquigarrow$ is strongly normalizing on the well-typed CNIC terms.*

CONJECTURE 2 (Consistency). *If Σ contains no modal variables, then there is no term M such that $\Sigma;\cdot \vdash M : \Pi A\!:\!\mathsf{Prop}\,.\,A$*

4. The `cinic` Implementation

Our initial implementation of CNIC is called `cinic`, written in approximately 6kloc in OCAML. Figure 4 presents the concrete syntax of `cinic`, in relation to the abstract syntax. The context `G` in `fun`- and `nfun`-terms is called the *parameter context*. It lists parameters which may be mentioned in the type of the scrutinee. The expression `e` in the syntax for `fun`- and `nfun`-terms is the expected type for the `fun`- or `nfun`-term. The different kinds of cases a for `nfun`-terms are for names x in the context, the N'th variable in the nested ν-abstraction against which the `nfun` is matched, or, finally, a catch-all case matching any other name x.

		`(Type i)`	$\sim$	$\texttt{Type}_i$
		`(Name e)`	$\sim$	$\texttt{Name}\, e$
		`(\ x e e')`	$\sim$	$\lambda x : e.e'$
		`(! x e e')`	$\sim$	$\Pi x : e.e'$
		`(e1 ... en)`	$\sim$	$(e_1 \ldots e_n)$
		`(nu x e e')`	$\sim$	$\nu\, x{:}e\,.\,e'$
		`(^ x e e')`	$\sim$	$\nabla x{:}e\,.\,e'$
		`(@ e x1 ... xn)`	$\sim$	$e\ @\ (x_1, \ldots, x_n)$
		`(fun x G e c1 ... cn)`	$\sim$	$\texttt{fun}\, x\, G\, e\, c_1\, \ldots c_n$
		`(nfun G e a1 ... an)`	$\sim$	$\texttt{nfun}\, G\, e\, a_1\, \ldots a_n$
		where:		
`G`	`::=`	`((x1 e1) ... (xn en))`	$\sim$	$(x_1 : e_1, \ldots, x_n : e_n)$
`c`	`::=`	`(d G e)`	$\sim$	$d\backslash G \to e$
`a`	`::=`	`(x e)`	$\sim$	$x\backslash\cdot \to e$
		`| (N e)`	$\sim$	$N\backslash\cdot \to e$
		`| (x G e)`	$\sim$	$x\backslash G \to e$

Figure 4. Concrete Syntax of `cinic`

We currently do not implement a number of useful features found in mature implementations of type theory, like Coq:

- We currently use an S-expression syntax for CNIC for easy parsing, and there is no syntax extension mechanism.
- We do not support implicit arguments.
- There is no tactic language, so proof terms are written directly.
- Type refinement in pattern cases is done just by matching, rather than unification.
- Inversion and injectivity lemmas, which could be derived automatically, currently must be proved by hand.

While future work includes improvements on all those points, the advantages of CNIC with regard to variable binding may still make even the current version of `cinic` attractive. The most burdensome limitation, in our opinion, is the need to derive inversion and injectivity by hand. The other issues are less costly to work around.

4.1 Commands

The `cinic` tool processes a sequence of commands one at a time. The central commands are `Inductive`, for declaring a set of mutually inductive types, and `Define`, for defining a constant to equal a given term. The syntax for these commands is:

```
(Inductive D1 ... Dn)
(Define x e' e)
(Define x e)

where

D ::= (x e C1 ... Cn)
C ::= (x e)
```

In `Define`-commands, the optional expression `e'` above is the expected type for `e`, which the `Define`-command defines `x` to equal. In `Inductive`-commands, a sequence of `D`-expressions define a set of mutually inductive types. In a `D`-expression, `x` is the name of the type constructor, `e` is its kind, and the following `C`-expressions define its term constructors `x` with their types `e`.

4.2 Implicit Contexts

One feature of `cinic` that helps in writing proof terms directly is support for *implicit contexts* in pattern cases. The basic idea is that the user need not specify pattern variables for cases of pattern-matching functions. Instead, `cinic` will introduce the pattern variables automatically, with canonical names. The benefit of this approach is that it reduces the number of names that the programmer must choose – and keep track of. Instead of keeping track of new names in each case, the programmer must just keep track of a single set of ways of forming canonical names.

The names for pattern variables which `cinic` automatically introduces with the implicit contexts feature consist of the name of the scrutinee concatenated with the name of the argument as listed in the type of the term constructor. For example, suppose we have a constructor `cons` for a standard inductive type for homogeneous polymorphic lists. This constructor has the type:

$$\Pi A : \texttt{Type}_0.\, \Pi a : A.\, \Pi r : (\texttt{list}\ A).\, (\texttt{list}\ A)$$

In the pattern case for `cons`, the programmer can certainly choose her own names for the pattern variables for these three arguments, listing these names in a context following “`cons`” at the start of the case. But she may also omit that context. In that case, `cinic` will automatically fill it in as follows, where x is the variable listed in the pattern-matching function's type for the scrutinee:

$$((x.A\ :\ \texttt{Type}_0)\ (x.a\ :\ x.A)\ (x.r\ :\ (\texttt{list}\, x.A)))$$

Here “$x.A$”, for example, is just the name of the variable which `cinic` introduces, not special syntax. In the body of the case, the programmer may refer to the arguments of the `cons`-term via these canonical names. We will see examples of implicit contexts below.

5. Confluence of Untyped Lambda Calculus

We have implemented in `cinic` a proof given by Barendregt (who attributes it without citation to Tait and Martin-Löf) of confluence of untyped lambda calculus, based on a multi-step reduction relation (also sometimes called simultaneous reduction [22]) [3]. The crux of the proof is to define a relation which is between (in the inclusion ordering) $\to_\beta$ and $\to_\beta^*$, and show that this relation satisfies the diamond property. To formalize this reasoning in `cinic`, we first declare an indexed inductive types `Step` for single step reduction and `Mstep` for multi-step reduction. We rely on a generic reflexive-transitive closure operator in `cinic`'s library to represent $\to_\beta^*$.

For the diamond property, we prove by induction on a term `dleft` of type `(Mstep t s1)` that whenever we additionally have a term `dright` of type `(Mstep t s2)`, we can construct a term of type `(Mstep_join s1 s2)`, where `Mstep_join` is constructed by exhibiting a term `t'` and derivations of `(Mstep s1 t')` and `(Mstep s2 t')`. To complete the proof, we prove that the relation is indeed between $\to_\beta$ and $\to_\beta^*$, and that this implies confluence of $\to_\beta$.

```
(Inductive (trm (Type 0)
   (var (! name (Name trm) trm))
   (app (! fn trm
        (! arg trm
         trm)))
   (lam (! body (^ n trm trm)
         trm))))
```

Figure 5. The Datatype for Terms

The total development is just under 700 lines of `cinic`, not counting around 225 lines of inversion lemmas, which in a more mature implementation would be derived automatically. We now walk through the development, in `cinic` concrete syntax, to see how the nominal features of CNIC are used in a non-trivial example like this one. We start with the basic datatypes and the definition of substitution, then consider the proof of the diamond property for `Mstep`, and finally look at the proof that this implies confluence of `Step`.

5.1 The `trm` Type

Terms of the untyped lambda calculus are encoded as elements of an inductive type `trm`, presented in Figure 5. For use with the implicit contexts feature described above, we choose descriptive names for the arguments to the constructors. The `var` constructor takes a single argument `name`, of type `(Name trm)`. Such names are introduced by the `lam` constructor, whose single argument `body` is a ∇-abstraction. Thus, object language binding will be implemented with ν-abstraction in the encoding, in accordance with our higher-order encoding methodology.

5.2 Substitution

Figure 6 gives the `cinic` code defining substitution. To substitute `t1` for `x` in a term `t2`, we would write `(subst t1 (nu x trm t2))` using this implementation. The `fun`-term in the definition thus scrutinizes terms of type `(^ x trm trm)`. The cases of this `fun`-term are written using implicit arguments. The context automatically constructed by `cinic` for the app-case, for example, is:

```
((t2.fn (^ x trm trm)) (t2.arg (^ x trm trm)))
```

The names of the pattern variables are derived as explained above, where `t2` has been identified as the name of the scrutinee from `subst`'s stated type `(! t2 (^ x trm trm) trm)`. Because the scrutinee's stated type `(^ x trm trm)` is a ∇-abstraction, `cinic` recognizes that we are matching beneath ν-abstractions, and hence must raise the types of all the pattern variables: the subterms of the scrutinee can all depend on those ν-bound variables. That is why the types for `t2.fn` and `t2.arg` are both `(^ x trm trm)`, and not just `trm`.

Notice that we need not worry about possible variable capture in our definition of `subst`, since CNIC ensures this cannot happen with ν-bound names. This is why the `lam`-case for `subst` is so much simpler than it would be if we were using a concrete encoding of names. So this is an example of the sort of situation where the nominal features of CNIC making programming or proving with binders much less burdensome.

The `var` case of the substitution uses a name-matching function to do a case split on whether this variable is the one for which we are substituting, or a distinct variable. Because we are operating beneath a ν-abstraction for x, this ends up being a case split on whether `t2.name` (whose type is raised, as explained just above) equals νx : `trm`.x, or νx : `trm`.y for some distinct name y. The first case of the `nfun`-term is for matching `(nu x trm x)`, and the second for matching `(nu x trm y)`, for any distinct name `y`.

```
(Define subst (! t1 trm
              (! t2 (^ x trm trm)
                 trm))
  (fun subst
      ((t1 trm))
      (! t2 (^ x trm trm) trm)
    (var
      ((nfun ()
        (! n2 (^ x trm (Name trm)) trm)
        (0 t1)
        (y ((y (Name trm))) (var y)))
       t2.name))
    (app
      (app (subst t1 t2.fn) (subst t1 t2.arg)))
    (lam
      (lam (nu n trm
             (subst t1
               (nu x trm (@ t2.body x n))))))))
```

Figure 6. Substitution

```
(Inductive
  (Mstep (! t1 trm (! t2 trm (Type 0)))
    (Mstep_var_refl
        (! name (Name trm)
          (Mstep (var name) (var name))))
    (Mstep_lam
        (! body1 (^ x trm trm)
        (! body2 (^ x trm trm)
        (! dbody (^ x trm
                   (Mstep (@ body1 x)
                          (@ body2 x)))
          (Mstep (lam body1) (lam body2))))))
    (Mstep_app1
        (! fn1 trm (! arg1 trm
        (! fn2 trm (! arg2 trm
        (! dfn (Mstep fn1 fn2)
        (! darg (Mstep arg1 arg2)
          (Mstep (app fn1 arg1)
                 (app fn2 arg2))))))))
    (Mstep_app2
        (! fn1 trm (! arg1 trm
        (! body2 (^ x trm trm) (! arg2 trm
        (! dfn (Mstep fn1 (lam body2))
        (! darg (Mstep arg1 arg2)
          (Mstep (app fn1 arg1)
                 (subst arg2 body2))))))))))
```

Figure 7. The Datatype for Multi-Step Reduction

5.3 Multi-Step Reduction

Figure 7 gives an inductive definition of the multi-step reduction relation. Variables step to themselves; lambda-abstractions step as their bodies do; and for applications, we can either step the functional subterm and argument subterm in parallel (`Mstep_app1`), or we can do so and then do a β-reduction (`Mstep_app2`). Note the use of name replacements in the `Mstep_lam` case, to state that the body of the first lambda-abstraction steps to the body of the second, when the bodies both use `x` as the name of the bound variable.

5.4 Multi-Step Reduction and Substitution Lemmas

The two lemmas about multi-step reduction needed in the proof are:

```
Mstep_refl :
   (! t trm (Mstep t t))
```

and

```
(Define Mstep_refl
  (fun IH()
        (! t trm (Mstep t t))
    (var
       (Mstep_var_refl t.name))
    (app
       (Mstep_app1 t.fn t.arg t.fn t.arg
          (IH t.fn) (IH t.arg)))
    (lam
       (Mstep_lam t.body t.body
          (nu x trm (IH (@ t.body x)))))))
```

Figure 8. Reflexivity of Multi-Step Reduction

```
Mstep_subst :
   (! s2 trm
   (! t2 trm
   (! d2 (Mstep s2 t2)
   (! s1 (^ x trm trm)
   (! t1 (^ x trm trm)
   (! d1 (^ x trm (Mstep (@ s1 x) (@ t1 x)))
     (Mstep (subst s2 s1) (subst t2 t1))))))))
```

The first lemma says that Mstep is reflexive for all terms (not just variables, as the definition of Mstep already states). The second lemma states that, using conventional notation for substitution, $[s_2/x]s_1$ (multi-step) reduces to $[t_2/x]t_1$ if s_2 reduces to t_2 and s_1 to t_1. The proofs of these lemmas are very short, just 55 lines. For example, the proof of the first lemma is listed in Figure 8. Note the use of implicit contexts to avoid having to invent names for all the pattern variables. We hope the reader agrees that this makes the proof of Mstep_refl quite readable.

The proof of Mstep_subst depends on a lemma describing how substitution commutes with itself. This well known property is written in conventional mathematical notation like this, where x is not free in t_1:

$$[s/y][t_1/x]t_2 \;=\; [[s/y]t_1/x][s/y]t_2$$

In our cinic development, this lemma is stated as follows, making use of a standard indexed inductive type eq for polymorphic equality (as in Coq):

```
subst_comm :
 (! s trm
 (! t1 (^ y trm trm)
 (! t2 (^ y trm (^ x trm trm))
  (eq trm
    (subst s (nu y trm
                (subst (@ t1 y) (@ t2 y))))
    (subst (subst s (nu y trm (@ t1 y)))
           (nu x trm
              (subst s (nu y trm (@ t2 y x)))))))))
```

The proof, by induction on t2, and is 87 lines long, and requires in one of the variable cases this lemma, which states that $[s/x]t = t$ when x is not free in t:

```
subst_closed :
  (! s trm
  (! t trm
    (eq trm (subst s (nu x trm t)) t)))
```

The proof of that lemma is 31 lines long. In systems based on HOAS, where object language substitution is mapped to meta-language substitution, these two lemmas about substitution would not be required. In the CNIC methodology, object language substitution must be implemented, but can be done so in a straightforward way using the nominal features of the language. The resulting required lemmas are also straightforward to prove, without the low-level lemmas (e.g. about shifting, for de Bruijn indices) that are typically required with concrete encodings of variable names.

5.5 The Diamond Property for Mstep

The diamond property for Mstep is now formulated as follows. First, we make this definition:

```
(Define P (\ t trm
          (\ s1 trm
          (! s2 trm
          (! dright (Mstep t s2)
            (join s1 s2))))))
```

This defines a predicate P, for which we can prove by induction on dleft:

```
Mstep_confl :
  (! t trm
  (! s1 trm
  (! dleft (Mstep t s1)
    (P t s1))))
```

This proof is non-trivial, requiring around 275 lines of cinic. The most significant challenge is that we must apply inversion in numerous cases. For example, in the case where dleft is built with Mstep_lam, cinic's type refinement will refine the trm t to be (lam dleft.body1), and s1 to be (lam dleft.body2). But it is up to us to apply an inversion lemma to conclude that since t is a lam-trm, the proof dright must also be built with Mstep_lam (since that is the only proof which allows a lam-trm to take a multi-step), and s2 must then also be a lam-trm. As remarked above, the current implementation of cinic does not derive this inversion lemma automatically, as Coq, for example, does. Instead, the programmer must currently derive such principles himself.

5.6 Concluding Confluence of Step

Figure 9 gives an inductive definition of the single-step reduction relation. Figure 10 gives the inductive definition from cinic's library of the reflexive transitive closure star of a relation. Using star, we can now define the desired new reduction relation, Starstep:

```
(Define Starstep (star trm Step))
```

Given the proof of the diamond property for Mstep, it is possible to derive a proof of the diamond property for Starstep which is equivalent to a proof of confluence of Step.

Informally, the proof is as follows. Assuming for two relations, R and S, $R \subseteq S \subseteq R^*$ and the diamond property for S, confluence for R can be derived as follows, in three major steps. First, assume $R^*\, t\, s_1$ and $R^*\, t\, s_2$. From this get $S^*\, t\, s_1$ and $S^*\, t\, s_2$ by monotonicity of $*$, using $R \subseteq S$. Second, by confluence of S, which follows easily from the diamond property for S, get a t' with $S^*\, s_1\, t'$ and $S^*\, s_2\, t'$. Finally, get $R^{**}\, s_1\, t'$ and $R^{**}\, s_2\, t'$ again by monotonicity of $*$, this time using $S \subseteq R^*$. Confluence of R can be concluded from this via idempotence of *.

Now we consider how this is implemented in cinic. For convenience, we first define StarMstep:

```
(Define StarMstep (star trm Step))
```

From cinic's library, we have a lemma star_commute, which states that star maintains commutativity:

```
(! A (Type 0)
(! R1 (! x A (! y A Prop))
(! R2 (! x A (! y A Prop))
(! t1 A
(! s1 A
(! s2 A
(! d1 ((star A R2) t1 s1)
(Inductive
   (Step (! t1 trm (! t2 trm Prop))
      (Step_beta
          (! body (^ x trm trm)
          (! arg trm
            (Step (app (lam body) arg)
                  (subst arg body)))))
      (Step_app1
          (! fn1 trm
          (! fn2 trm
          (! arg trm
          (! d (Step fn1 fn2)
            (Step (app fn1 arg)
                  (app fn2 arg)))))))
      (Step_app2
          (! fn trm
          (! arg1 trm
          (! arg2 trm
          (! d (Step arg1 arg2)
            (Step (app fn arg1)
                  (app fn arg2)))))))
      (Step_lam
          (! body1 (^ x trm trm)
          (! body2 (^ x trm trm)
          (! d (^ x trm (Step (@ body1 x)
                              (@ body2 x)))
            (Step (lam body1)
                  (lam body2))))))))
```

Figure 9. The Datatype for Single-Step Reduction

```
(Inductive
   (star (! A (Type 0)
         (! R (! x A (! y A Prop))
          (! a A (! b A Prop))))
      (star_refl
          (! A (Type 0)
          (! R (! x A (! y A Prop))
            (! a A (star A R a a)))))
      (star_next
          (! A (Type 0)
          (! R (! x A (! y A Prop))
          (! a A
          (! b A
          (! c A
          (! d1 (R a b)
          (! d2 (star A R b c)
            (star A R a c))))))))))
```

Figure 10. Inductive Definition of Reflexive Transitive Closure

```
(! d2 ((star A R1) t1 s2)
(! d3 (commutativity A R1 R2)
 (join A (star A R1) (star A R2) s1 s2))))))))))
```

To prove confluence for Mstep, the proof of the diamond property for Mstep is supplied to star_commute, along with StarMstep for both R1 and R2. The final type for the proof of confluence of Mstep is shown below:

```
(Define Mstep_confl
    (! t1 trm
    (! s1 trm
    (! s2 trm
    (! d1 (StarMstep t1 s1)
    (! d2 (StarMstep t1 s2)
      (join trm StarMstep StarMstep s1 s2)))))))
```

The final step of our informal proof is to show that Step $\subseteq$ Mstep $\subseteq$ Starstep. We formalize this with two lemmas. The first, step_to_mstep, is an easy conversion between Step and Mstep. The second, mstep_to_starstep, is a conversion between Mstep and Starstep. This conversion is less trivial in that congruence properties must be derived inductively for Starstep. The type of one such congruence property is:

```
(! body1 (^ n trm trm)
(! body2 (^ n trm trm)
(! d (^ n trm
       (Starstep (@ body1 n) (@ body2 n)))
  (Starstep (lam body1) (lam body2)))))
```

Now that we have confluence of Mstep and the inclusion relations between Step, Mstep, and Starstep, we can follow the steps laid out in the informal proof in a straightforward manner. The formal proof first assumes (Starstep t s1) and (Starstep t s2). Recall from the informal proof that these assumptions must first be converted from type Starstep to type StarMstep. As was done in the informal proof, the formal proof does this via monotonicity of star, implemented in cinic's library as a lemma star_mono:

```
(! A1 (Type 0)
(! A2 (Type 0)
(! R1 (! x A1 (! y A1 Prop))
(! R2 (! x A2 (! y A2 Prop))
(! f (! x A1 A2)
(! q (! s A1 (! t A1
     (! d (R1 s t)
         (R2 (f s) (f t)))))
(! s A1
(! t A1
(! d (star A1 R1 s t)
  (star A2 R2 (f s) (f t))))))))))))
```

Combining star_mono with step_to_mstep, produces another conversion lemma, starstep_to_starmstep, with the type expected from its name. The next step in the informal proof is to derive a joining point and two joining relations. In the formal proof this is as easy as the application of the Mstep_confl proof. Recall, however, that this proof returns all three parts as a join bundle. To separate them, the join has to be inverted which is handled easily by the invert_join lemma from cinic's library. Once they are separated, the final step of the proof, converting the output relations, can be completed.

Just as in the first step, this conversion is a basic application of star_mono and a conversion lemma, mstep_to_starstep. These two alone produce a relation of type (star trm (Starstep)) which can be reduced to Starstep via idempotence of star as proved by the star_idem lemma from cinic's library:

```
(! A (Type 0)
(! R (! x A (! y A Prop))
(! s A
(! t A
(! d (star A (star A R) s t)
     (star A R s t))))))
```

Combining all three of these lemmas produces a new conversion lemma, starmstep_to_starstep, with the expected type. Once the output relations are attained, all that remains is to rejoin them with the joining point which is handled simply by a join constructor, show_join, in cinic's library.

Putting all of the pieces together, the final formulation of confluence of Step is shown in Figure 11. In total, the proof of confluence for Step, including lemmas, given the diamond property for Mstep, requires only around 175 lines of code, not including library or inversion lemmas.

```
(Define Step_confl
    (\ t1 trm
    (\ s1 trm
    (\ s2 trm
    (\ d1 (Starstep t1 s1)
    (\ d2 (Starstep t1 s2)
  (invert_join trm StarMstep StarMstep s1 s2
    (Mstep_confl t1 s1 s2
      (starstep_to_starmstep t1 s1 d1)
      (starstep_to_starmstep t1 s2 d2))
  (join trm Starstep Starstep s1 s2)
      (\ joining trm
      (\ d1 (StarMstep s1 joining)
      (\ d2 (StarMstep s2 joining)
        (show_join trm Starstep Starstep
                   s1 s2 joining
          (starmstep_to_starstep s1 joining d1)
          (starmstep_to_starstep s2 joining d2)
  )))))
))))))
```

Figure 11. Final Proof of Confluence for `Step`

5.7 A Note on Extensionality

The definitional equality of CNIC follows CIC in being intensional: extensionally equal functions are not necessarily definitionally equal. The same is true in CNIC for terms of ∇-type. We do not generally have, for t of type $\nabla x : A \,.\, B$, that t is definitionally equal to $\nu\, x : A \,.\, t\ @\ x$. While we conjecture that CNIC's type refinement could be modified to imply extensionality when B is an inductive type, currently we postulate extensionality at ∇-types as an axiom. The confluence proof uses such a principle of extensionality to obtain this equation between encoded lambda-terms: `lam t` = `lam (nu x trm (@ t x))`, which is needed in several places. Extensionality at ∇-type is the only axiom we add to CNIC for the confluence proof.

6. Related Work

The most well-known intensional approach to encoding name-binding is Higher-Order Abstract Syntax, or HOAS, which refers to the use of λ-abstractions to encode binding constructs. HOAS has been combined successfully with logic programming in Twelf [17], λProlog [12], and Bedwyr [2], where the latter is the source of the ∇-abstraction. HOAS has proved difficult to combine with functional programming languages, however, which include type theories such as CIC, because it is difficult to express recursion over λ-abstractions [10, 6, 25]. A number of systems have addressed this problem [21, 18], including the ∇-calculus that is the source of the ν-abstraction, but these require complex machinery to recurse over λ-abstractions (this is the purpose of the ν-abstraction in the ∇-calculus), and none of these approaches have been shown compatible with polymorphism and thus with CIC.

The most closely related approach to the current work is Simple Nominal Type Theory, or SNTT [4], which contains constructs that directly mirror the ∇-type, ν-abstractions, and name replacements. The current work can be seen as an extension of this work to handle the full language of CIC.

There are also a number of extensional approaches to encoding name-binding [7, 19, 11, 8]. The most well-known of these is Nominal Logic [7], in which bindings are encoded as α-equivalence classes. Nominal Logic has been successfully combined with the Isabelle proof assistant [24], which is based on extensional type theory. There has been one attempt to combine Nominal Logic [1] and the Theory of Context [11] with CIC, but the first work (by its own admission) states that it is incomplete and difficult to use, and both require `Axiom` statements. Further, most of these approaches do not satisfy the typing property of name-bindings discussed in Section 2.

Another interesting extensional approach is the Locally Nameless approach, in which bound names are encoded with deBruijn indices while free names are encoded with a known set of names [9]. This approach has been shown to be very easy to use, as all operations related to deBruijn indices can be encapsulated in a small term manipulation library. Unfortunately, deBruijn indices do not satisfy the scoping property, as there is nothing to ensure that only valid numbers are used. For example, the term `var\;200` is only valid under 200 binders. Thus the Locally Nameless approach is not truly an adequate encoding, as discussed above, without an extra judgment that the bound variables in an expression are all valid. All operations on expressions must thus be proved to correctly manipulate names as well, drastically increasing the size of proofs. In addition, this approach does not satisfy the typing property.

7. Conclusion

This paper has presented a theory, the Calculus of Nominal Inductive Constructions, for encoding and reasoning about name-bindings. This calculus is an extension of the Calculus of Inductive Constructions to include an intensional account of name-binding in terms of a construct called the ν-abstraction. CNIC also includes powerful features for using name-bindings, including name-matching functions for comparing names and pattern-matching functions that can match inside ν-abstractions. CNIC has been demonstrated with a modest-sized example, a proof of confluence of the untyped λ-calculus, which relies heavily on name-binding. This proof has been machine-checked with an implementation of CNIC called `cinic`.

There are many interesting directions for future work on CNIC. It would be useful in CNIC to add support for dependent pattern-matching with stronger type refinements, such as initially suggested by Coquand [5]. In standard type theory, this dependent pattern-matching is known to require the axiom of Uniqueness of Identity Proofs (UIP). In CNIC, however, this type refinement would have to work for pattern-matches inside ν-abstractions, which seems to require additional axioms, such as extensionality of ν.

On the theoretical side, there are many interesting theorems in CNIC that intuitively should hold but do not seem to be provable in the theory. For example, if x, y, and their type B are fresh for α, then the proposition $\nabla\alpha : A \,.\, \mathtt{eq}\ B\ x\ y$ should intuitively imply that $\mathtt{eq}\ B\ x\ y$ holds, as the equality proof should not require α. Proving this is not immediate, however, and we conjecture that it cannot in fact be proved in CNIC. This suggests that there are non-trivial models of CNIC, and thus of name-binding, which are highly counter-intuitive.

References

[1] B. Aydemir, A. Bohannon, and S. Weirich. Nominal reasoning techniques in Coq (extended abstract). In *Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006)*, pages 69–77, 2007.

[2] D. Baelde, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and École Polytechnique. The Bedwyr system for model checking over syntactic expressions. In *21st Conference on Automated Deduction (CADE '07)*, pages 391–397, 2007.

[3] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1984.

[4] J. Cheney. Simple nominal type theory. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[5] T. Coquand. Pattern matching with dependent types. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical frameworks*, 1992.

[6] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL '96)*, pages 284–294, 1996.

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

[8] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS '99)*, 1999.

[9] C. McBride and J. 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, 2004.

[10] E. Meijer and G. Hutton. Bananas in Space: Extending fold and unfold to Exponential Types. In *Proceedings of the 7th SIGPLAN-SIGARCH-WG2.8 International Conference on Functional Programming and Computer Architecture*. ACM Press, La Jolla, California, June 1995.

[11] M. Miculan. Developing (meta)theory of λ-calculus in the theory of contexts. In *Proceedings of the Workshop on MEchanized Reasoning about Languages with variable bINding (MERLIN '01)*, pages 65–81, 2001.

[12] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

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

[14] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] M. Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 48–59, 1994.

[16] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[17] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[18] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[19] U. Schöpp and I. Stark. A dependent type theory with names and binding. In *Computer Science Logic (CSL '04)*, volume 3210 of *LNCS*, pages 235–249, 2004.

[20] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001.

[21] C. Schürmann and A. Poswolsky. Practical programming with higher-order encodings and dependent types. In *17th European Symposium on Programming (ESOP '08)*, pages 93–107, 2008.

[22] TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[23] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0*, 2004. http://coq.inria.fr.

[24] C. Urban. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4), 2008.

[25] G. Washburn and S. Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the 8th International Conference on Functional Programming (ICFP '03)*, pages 249–262, 2003.

[26] E. Westbrook. *Higher-Order Encodings with Constructors*. PhD thesis, Washington University in Saint Louis, 2008.