Toward a General Theory of Names, Binding and Scope

James Cheney

University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Abstract

High-level formalisms for reasoning about names and binding such as de Bruijn indices, various flavors of higher-order abstract syntax, the Theory of Contexts, and nominal abstract syntax address only one relatively restrictive form of scoping: namely, unary lexical scoping, in which the scope of a (single) bound name is a subtree of the abstract syntax tree (possibly with other subtrees removed due to shadowing). Many languages exhibit binding or renaming structure that does not fit this mold. Examples include binding transitions in the π-calculus; unique identifiers in contexts, memory heaps, and XML documents; declaration scoping in modules and namespaces; anonymous identifiers in automata, type schemes, and Horn clauses; and pattern matching and mutual recursion constructs in functional languages. In these cases, it appears necessary to either rearrange the abstract syntax so that lexical scoping can be used, or revert to first-order techniques.

The purpose of this paper is to catalogue these "exotic" binding, renaming, and structural congruence situations; to argue that lexical scoping-based syntax techniques are sometimes either inappropriate or incapable of assisting in such situations; and to outline techniques for formalizing and proving properties of languages with more general forms of renaming and other structural congruences.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Languages

Keywords names, abstract syntax, scope, nominal logic

1. Introduction

It is a significant challenge to mechanize reasoning about syntactic structures involving bound names (e.g., abstract syntax trees representing the terms or formulas of a programming language or logic). It is widely agreed that to perform such reasoning using a "primitive", or first-order, syntax encoding is too painful to contemplate for anything as complex as a full-scale programming language. A wide variety of techniques for automatically managing the tedious details attendant upon formalizations of abstract syntax with bound names have been proposed. These include name-free approaches such as combinators and de Bruijn representations [7] as well as higher-order approaches such as higher-order abstract syntax [20], weak higher-order abstract syntax [8], and lambda-term abstract syntax [14]. Another recently proposed technique is the approach of Gabbay and Pitts [10], which focuses on alpha-equivalence axiomatized in terms of name-swapping and freshness. Additional techniques such as Hybrid [18], the Theory of Contexts [11], and $FO\lambda^{\Delta\nabla}$ [16] have also recently been proposed. In this paper we employ *nominal abstract syntax*, a simplified form of *nominal logic* [21].

Scope is a fundamental concept when discussing binding. If we view a syntax representation as an abstract data structure, then the scope of a binding occurrence (located at position p) of an identifier x is the set of positions in the tree at which a reference to x refers to *the* x bound at p.

All of the above techniques have a common limitation. They provide advanced support for only one kind of binding/scoping behavior: *unary lexical scoping*, in which the abstract syntax is tree-structured, one name is bound at a time, and its scope is a subtree of the abstract syntax tree (possibly with subtrees removed due to shadowing). Unary lexically scoped (ULS) binding is exhibited by, for example, the λ-calculus's $\lambda x.t$ term constructor, logical quantifiers, and a wide variety of other forms of binding. While this is quite an important and flexible form of binding, it is not the only one encountered frequently in real programming languages, operational semantics rules, or logics. For example,

1. In $let\ x = e\ in\ e'$, the scope of x is e', but not e.
2. In π-calculus "binding" transitions such as $p \xrightarrow{x(y)} q$ and $p \xrightarrow{\bar{x}(y)} q$, y is bound in q.
3. In a C program, a global (`extern`) definition of an identifier is considered bound within the whole program; such identifiers can be defined at most once. Similarly, in C, a local (`static`) defining occurrence has file scope, so must be unique within its file, but may be reused in other files.
4. In module or namespace systems (e.g. Wells and Vestergaard's module calculus [29]), an identifier can usually be defined at most once within a given scope; also, toplevel declarations may have sequential scope (e.g. in ML) or global scope (e.g. in Haskell or C/C++). In addition, modules often exhibit "open" scoping, where identifiers defined in one module may be referenced in another module, with the binding resolved at link-time rather than compile-time.
5. In XML [1], `ID` attribute values are required to be globally unique "definitions" of an identifier, and `IDREF` attributes must refer to `ID`s. `IDREF`s may refer to `ID`s in physically different files storing different parts of an XML document.

MERLIN'05 September 30, 2005, Tallinn, Estonia.

6. Finite automata are often considered equal up to one-to-one renaming of state names [12]. Similarly, Standard ML type schemes [17] and Horn clauses in logic programs [6] are often considered to be equal up to one-to-one renaming of free variables. Thus, it is always safe to assume that the identifiers in an automaton, Horn clause, or type scheme are completely fresh.
7. In pattern-matching expressions in functional languages (e.g. Haskell or SML), the free variables of the pattern are considered bound simultaneously in the body.
8. In a mutually recursive definition (e.g. LISP or Scheme's `letrec`), the identifiers being defined are considered in scope in all the definition bodies.

In this paper we argue that unary lexical scoping is often either inconvenient or inadequate for encoding interesting forms of scoping encountered in real programming or formalization situations. In the former case, the techniques can still be used, but the language encoding must be contorted to accommodate ULS. Making these forms of binding fit the mold of unary lexical scoping complicates the syntax significantly, so that the form and structure of paper proofs and their formalizations diverge. In the latter case, standard binding techniques can be used, but do not provide an *adequate* encoding of the syntax: that is, there may be "confusion" concerning which object term corresponds to a representation, or "junk" terms in the representation type not corresponding to any object term. When adequacy fails, it is necessary to fall back on low-level reasoning techniques. We believe that this lack of expressiveness is a significant obstacle to adoption of formal methods for reasoning about programming languages by non-experts.

Although the name-abstraction operation provided by nominal logic shares the limitations of the other techniques, we argue that nominal logic is flexible enough to provide better support for forms of binding beyond unary lexical scoping, and including all of the examples listed above. This paper represents work in progress, and despite our emphasis on nominal techniques, is not meant to say anything definitive about the advantages or disadvantages of the technique we propose over other approaches, but only to encourage discussion of new ideas and directions for all techniques for reasoning about binding.

The structure of the rest of the paper is as follows. In the next section, we provide a brief overview of nominal abstract syntax. The next four sections present four forms of binding and scoping for which unary lexical scoping is either inconvenient or inadequate, and show how the desired behavior can be formalized in nominal logic. In Section 3 we consider *pseudo-unary lexical scoping* forms such as let, typed quantifiers, and binding transitions in the π-calculus. These forms of binding are "almost" lexical, so can be handled using ULS, but doing so requires reorganizing the language, sometimes to the detriment of clarity or modularity. In Section 4, we consider *global* scoping as exhibited in C, assembly language, XML `ID`s/`IDREF`s, and module systems, in which an identifier may be bound ("defined") at most once, but may be referred to anywhere in a collection of modules. In Section 5, we consider *anonymous scoping* as exhibited by state names in automata and free variables in logic programming Horn clauses or ML type schemes. In Section 6, we consider situations in which the names in a general data structure (such as a pattern matching expression) are considered bound in another subterm. Section 7 shows how the ideas introduced in the previous sections can be combined to handle the binding structure of mutual recursion (i.e., `letrec`). Section 8 discusses future work and Section 9 concludes.

2. Background

In this section we provide a brief overview of nominal abstract syntax. Much more detail can be found in the papers [21, 28, 9, 4].

Let $\mathbb{A}$ be a countably infinite set of names $\{\mathsf{a}, \mathsf{b}, \mathsf{x}, \mathsf{y}, \ldots\}$. The (ground) nominal terms are as follows:

$$t ::= c \mid f(\vec{t}) \mid \mathsf{a} \mid \langle \mathsf{a} \rangle t$$

Intuitively, the term $\langle \mathsf{a} \rangle t$ is considered to be an "abstraction", or a term with a distinguished bound name a; we will equip nominal terms with an equational theory that identifies abstractions up to alphabetic renaming (α-equivalence). Note that names are not variables; instead they are to be thought of as "special constants".

Nominal terms are classified using types. Types include base *name types* ν and *data types* δ, as well as a binary type constructor $\langle \nu \rangle \tau$ called *abstraction*, that combines a name type and ordinary type. Additional type constructors, in particular Cartesian products $\tau \times \tau'$ and lists $list\ \tau$, may also be present and are standard. Terms are considered well-formed (relative to some signature Σ assigning types to constants, names, and function symbols) as follows:

$$\frac{\mathsf{a} : \nu \in \Sigma}{\mathsf{a} : \nu} \quad \frac{c : \tau \in \Sigma}{c : \tau} \quad \frac{\mathsf{a} : \nu \quad t : \tau}{\langle \mathsf{a} \rangle t : \langle \nu \rangle \tau}$$

$$\frac{t_i : \tau_i \quad f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma}{f(\vec{t}) : \tau}$$

We define several relations and operations on ground nominal terms as follows. First, let π be a finite, type-preserving permutation on names; that is, an invertible function that moves at most finitely many names and such that if $\mathsf{a} : \nu$ then $\pi(\mathsf{a}) : \nu$. Some examples include the identity function id, transpositions $(\mathsf{a}\ \mathsf{b})$ that exchange two names (of the same type), and compositions $\pi \circ \pi'$; obviously, name-permutations form a group, and this group is generated by the pairwise transpositions of names of compatible types. We write $\pi \cdot t$ for the result of renaming all the names occurring in t according to π; this is calculated as follows.

$$\begin{aligned} \pi \cdot \mathsf{a} &= \pi(\mathsf{a}) \\ \pi \cdot c &= c \\ \pi \cdot f(\vec{t}) &= f(\pi \cdot \vec{t}) \\ \pi \cdot \langle \mathsf{a} \rangle t &= \langle \pi \cdot \mathsf{a} \rangle \pi \cdot t \end{aligned}$$

Note that permutations essentially ignore the tree and binding structure of terms and simply rename all occurrences of names (both "free", "bound", and "binding"). Note also that ordinary constants are fixed by permutations, whereas names can be renamed; this is what makes names different from ordinary constants.

Next, we define what it means for a name to be independent of (or *fresh* for) a term. Intuitively, a name a is fresh for a term t (that is, $\mathsf{a} \# t$) if t possesses no occurrences of a unenclosed by an abstraction of a. We define this using the following inference rules:

$$\frac{(\mathsf{a} \neq \mathsf{b})}{\mathsf{a} \# \mathsf{b}} \quad \frac{}{\mathsf{a} \# c} \quad \frac{\mathsf{a} \# t_i \quad (i = 1, \ldots, n)}{\mathsf{a} \# f(t_1, \ldots, t_n)}$$

$$\frac{\mathsf{a} \# \mathsf{b} \quad \mathsf{a} \# t}{\mathsf{a} \# \langle \mathsf{b} \rangle t} \quad \frac{}{\mathsf{a} \# \langle \mathsf{a} \rangle t}$$

Finally, we define an appropriate equality relation on nominal terms that identifies abstractions up to "safe" renaming.

$$\frac{}{\mathsf{a} \approx \mathsf{a}} \quad \frac{}{c \approx c} \quad \frac{t_i \approx u_i \quad (i = 1, \ldots, n)}{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)}$$

$$\frac{\mathsf{a} \approx \mathsf{b} \quad t \approx u}{\langle \mathsf{a} \rangle t \approx \langle \mathsf{b} \rangle u} \quad \frac{\mathsf{a} \# (\mathsf{b}, u) \quad t \approx (\mathsf{a}\ \mathsf{b}) \cdot u}{\langle \mathsf{a} \rangle t \approx \langle \mathsf{b} \rangle u}$$

The above system is essentially the equational theory of nominal logic (an extension of first-order logic with a theory of names,

binding, freshness, α-equivalence, and quantification over fresh names). In the rest of this paper, we will consider extensions to the above inference rules, typically in the form of rules for well-formedness freshness and equality specialized to a particular type constructor or term structure. These inference rules can be formalized within nominal logic as well. For presentation purposes, however, we will work in terms of inference rules.

3. Pseudo-Unary Scoping

One aspect of unary binding that is theoretically unproblematic, but unsatisfying in practice, is its treatment of binding constructs of the following forms:

$$let\ x = e_1\ in\ e_2$$
$$\forall x{:}\tau.\phi$$
$$\lambda x{:}\tau.e$$

In each case, the bound variable x is separated from its scope by a type or term in which occurrences of x are not considered bound. That is,

$$let\ x = f(x)\ in\ e_2(x) \approx_\alpha let\ y = f(x)\ in\ e_2(y)$$

This "natural" way of writing the expression does not match up with unary lexical scoping. Another example of this is in the form of binding exhibited by π-calculus labelled transitions:

$$p \xrightarrow{x(y)} q(y) \quad \approx_\alpha \quad p \xrightarrow{x(z)} q(z)$$
$$p \xrightarrow{\bar{x}(y)} q(y) \quad \approx_\alpha \quad p \xrightarrow{\bar{x}(z)} q(z)$$

In nominal logic (or indeed any formalism providing unary lexical scoping), we may represent such syntax correctly by rearranging the arguments so that the bound name is adjacent to its scope:

$$\begin{aligned} let\_exp &: (exp, \langle id\rangle exp) \to exp. \\ forall\_prop &: (ty, \langle id\rangle prop) \to prop. \\ lambda\_exp &: (ty, \langle id\rangle exp) \to exp. \\ in\_trans &: (proc, id, \langle id\rangle proc) \to trans. \\ bound\_out\_trans &: (proc, id, \langle id\rangle proc) \to trans. \end{aligned}$$

For the first three cases, this is no great burden, but the π-calculus transitions no longer factor into a process, action, and process. Some improvement is possible, for example by employing two kinds of transitions, relating the input process to either an output process or a process with a bound name (see e.g. various encodings by Miller and Tiu [13, 15, 27]). However, this technique departs from the original description of the π-calculus.[1]

In nominal logic, however, nothing prevents us from adding axioms that directly describe other forms of binding than plain-vanilla abstraction. For example, if we take

$$let\_exp \quad : \quad (id, exp, exp) \to exp.$$

we may axiomatize the binding behavior by adding the following inference rules:[2]

$$\frac{\mathsf{x} \# e_1}{\mathsf{x} \# let\_exp(\mathsf{x}, e_1, e_2)}$$
$$\frac{\mathsf{x} \# f_2 \quad e_1 \approx f_1 \quad e_2 \approx (\mathsf{x}\,\mathsf{y}) \cdot f_2}{let\_exp(\mathsf{x}, e_1, e_2) \approx let\_exp(\mathsf{y}, f_1, f_2)}$$

[1] Of course, one may argue, as Miller and Tiu have done, that this change is an improvement.

[2] Here and elsewhere, we make use of the following theorem of nominal logic: if $\mathsf{x} \# e$ and $e = (\mathsf{x}\,\mathsf{y}) \cdot e'$, then $\mathsf{y} \# e'$. To see why this is the case, note that $\mathsf{x} \# e$ implies $(\mathsf{x}\,\mathsf{y}) \cdot \mathsf{x} \# (\mathsf{x}\,\mathsf{y}) \cdot e$, whence $\mathsf{y} = (\mathsf{x}\,\mathsf{y}) \cdot \mathsf{x} \# (\mathsf{x}\,\mathsf{y}) \cdot e = (\mathsf{x}\,\mathsf{y}) \cdot (\mathsf{x}\,\mathsf{y}) \cdot e' = e'$.

Similar rules can be written for $\forall$ and λ. For the π-calculus, if we use the signature

$$\begin{aligned} bout &: (id, id) \to act. \\ bin &: (id, id) \to act. \\ trans &: (proc, act, proc) \to trans. \end{aligned}$$

then the binding behavior can be axiomatized as

$$\frac{\mathsf{y} \# (p, \mathsf{x})}{\mathsf{y} \# trans(p, bout(\mathsf{x}, \mathsf{y}), q)}$$
$$\frac{p \approx p' \quad \mathsf{x} \approx \mathsf{x}' \quad \mathsf{y} \# q' \quad q \approx (\mathsf{y}\,\mathsf{y}') \cdot q'}{trans(p, bout(\mathsf{x}, \mathsf{y}), q) \approx trans(p', bout(\mathsf{x}', \mathsf{y}'), q')}$$

and similarly for bin. This captures the binding behavior of the π-calculus as originally presented.

These axioms do ensure that our user-defined binding constructs are treated correctly, but seem to require modifying the underlying equational and freshness theory on an ad hoc basis. The inference rules for equality above seem to have a very specific form, namely, we test that the parts outside the scope are equal, test that the bound name is fresh for the scope on the other side, then test that the scoped parts are equal up to renaming. We believe that it would be possible and worthwhile to find a more compact representation of scoping rules for such custom binding forms, from which the rules above can be extracted automatically. (One interesting possibility which has recently been proposed by Pottier is the `inner` and `outer` scope description keywords in Cαml [22], a tool which translates a high-level type specification to low-level OCaml code that deals with name binding automatically).

3.1 Sequential scoping

Another common form of scoping that is representable (but inconveniently so) using unary lexical scoping is what we shall term *sequential scoping*. Intuitively, an identifier has sequential scope if it occurs in a data structure that is part of a sequence of similar data structures, and it becomes bound in later elements of the sequence. Sequential scoping is "open-ended": for example, if we concatenate two sequences, then free references in the latter sequence may become bound to occurrences in the former.

In ML, for example,

```
fun f x = e_1
fun g x = e_2
fun f x = e_3
```

is allowed, and the first definition of f is shadowed by the second. This means that occurrences of f in e_1 and e_2 refer to the first definition, but occurrences of f in e_3 and later in the program will refer to the second definition.

We can handle sequential scoping using ordinary unary scoping by rearranging the abstract syntax. For example, we can express the above declarations as

$$fndecl(\langle \mathsf{f}\rangle([\mathsf{x}], e_1, fndecl(\langle \mathsf{g}\rangle([\mathsf{x}], e_2, fndecl(\langle \mathsf{f}\rangle([\mathsf{x}], e_3, nil))))))$$

where

$$fndecl : (\langle id\rangle(list\ id \times exp \times decllist)) \to decllist$$

But this has the obvious disadvantages of requiring us to use customized declaration lists. Can we do better?

We can axiomatize the desired behavior in nominal logic as follows:

$$\frac{f \# xs}{\mathsf{f} \# (fndecl(\mathsf{f}, xs, e)) :: ds}$$
$$\frac{xs \approx ys \quad \mathsf{f} \# (e', ds') \quad (e, ds) \approx (\mathsf{f}\,\mathsf{g}) \cdot (e', ds')}{fndecl(\mathsf{f}, xs, e) :: ds \approx fndecl(\mathsf{g}, ys, e') :: ds'}$$

Intuitively, this says that a function name is bound in its function body, and in any later declarations.

4. Global Scoping

Many situations, from XML documents to module systems to object files, employ a more primitive form of *global scoping with unique definitions*. Not only is the scope of a definition "global" within its original data structure, but other data structures may contain references to the identifier which become resolved later (for example, by a linker). In all of these situations, it is commonplace for a component (module, file, object, document fragment) to refer to names that are defined externally. In addition, references can be cyclic: that is, two modules may each define an identifier that the other uses. Similarly, in Separation Logic [23], a logic for reasoning about imperative programs that manipulate pointers, *heaps* are considered to be partial functions from identifiers to values that can be split into parts with disjoint domains (possibly with reference cycles crossing the two parts)

We wish to abstract away some of the above details in order to identify a core idea that may be incorporated into nominal logic and used for each of the above situations. In each situation, data structures can contain some names with special meaning: we call such names *defined* (without being too specific about what it means to define a name). Moreover, in any given data structure, names may be referenced as many times as desired, but may be defined at most once within a given scope; in particular, an operation combining two data structures that define names is only sensible when the sets of externally visible names defined in the structures are disjoint. Multiple data structures defining the same name may exist (think of several different versions of `libc`, each of which defines `malloc`), as long as they are not forced to coexist. On the other hand, uniquely defined identifiers can be hidden (for example, using the `static` keyword in C); that is, their scope can be limited so that the same identifier can be used in different components. As another example, parameter names in C function definitions must be distinct from each other but can be re-used in other scopes.

To model this behavior within nominal logic, we consider a new type constructor for *unique identifier definitions* (that is, a "uniqueness type"). We augment nominal terms and types with the following syntax:

$$t ::= \cdots \mid \mathsf{a}!\!! \qquad \tau ::= \cdots \mid \nu!\!!$$

The intended meaning of the unique name constructor is that a term or formula can be well-formed only if there are no duplicate unique names in it. We axiomatize this well-formedness property as a relation $S \vdash t$, where S is a superset of the uniquely defined names of term t. This generalizes the type system for ground nominal terms outlined in Section 2.1.

$$\frac{\mathsf{a} : \nu \in \Sigma}{S \vdash \mathsf{a} : \nu} \quad \frac{c : \tau \in \Sigma}{S \vdash c : \tau} \quad \frac{S \uplus \{\mathsf{a}\} \vdash t : \tau}{S \vdash \langle \mathsf{a} \rangle t : \langle \nu \rangle \tau} \quad \frac{\mathsf{a} : \nu \in \Sigma \quad \mathsf{a} \in S}{S \vdash \mathsf{a}!\!! : \nu}$$

$$\frac{S = \biguplus_1^n S_i \quad \bigwedge_{i=1}^n S_i \vdash t_i : \tau_i \quad f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma}{S \vdash f(t_1, \ldots, t_n) : \tau}$$

Note that S is a conservative upper bound on the set of names uniquely defined by t, so in the rule for abstraction, it is safe to add a to S to typecheck the body. Thus, $f(\mathsf{a}!\!!, \mathsf{a}!\!!)$ is ill-formed, while $f(\mathsf{a}!\!!, \mathsf{b}!\!!)$, $g(\mathsf{a}!\!!, \mathsf{a})$, and $f(\langle \mathsf{a} \rangle \mathsf{a}!\!!, \mathsf{a}!\!!)$ are well-formed. Using the uniqueness type, the identifier structure of XML documents can be modeled as follows:

$$\begin{aligned} id &: ident!\!! \to att. \\ idref &: ident \to att. \\ elt &: (string, list\ att, list\ elt) \to elt. \end{aligned}$$

Note that we distinguish between different sorts of attribute values at the type level, whereas in true XML, attribute values are just strings and type information is imposed externally by a DTD or XML Schema. In addition, since we are using a name type for identifiers, the string values of identifiers (which could, but frequently don't, carry interesting information in true XML) will be lost in our encoding. Also, in true XML, documents are considered equivalent up to reordering attributes within an element tag; although this could be captured with additional structural congruence axioms, or using a suitably axiomatized set type, we omit them for simplicity. Then the documents

```
<el1 id="id1">
  <el2 idref="id1"/>
</el1>
```

would be encoded as

$$elt(\texttt{"el1"}, [id(\mathsf{id}_1!\!!)], [elt(\texttt{"el2"}, [idref(\mathsf{id}_1)], [])])$$

whereas

```
<el1 id="id1">
  <el2 id="id1" idref="id1"/>
</el1>
```

has no (well-formed) encoding.

Similarly, defining occurrences of identifiers/parameters in C programs must be unique within their scopes, and this constraint can be enforced using $(-)!\!!$. Finally, we consider heaps and contexts. Such data structures are sometimes viewed as lists of pairs $list\ (id \times value)$, or alternatively, as finite, partial functions $id \to value$. Of course, these two encodings are not isomorphic, because the list representation is order sensitive and includes values such as $[(\mathsf{a}, x), (\mathsf{a}, y)]$ that correspond to no function. The first encoding emphasizes the concrete nature of such data structures, whereas the second emphasizes the fact that the identifiers being used as keys should appear at most once on the left-hand side. Using uniqueness types, we can combine the structural convenience of list processing with the unique mapping property guaranteed by the use of the function space by encoding heaps and contexts as values of type $list\ (id!\!! \times value)$. Of course, this representation comes with no built-in function application notation for lookups, but list lookup is not difficult to implement for this representation. Also, this representation is sensitive to ordering, but there are situations where this is desirable (e.g. "telescope" contexts as in dependent type theory).

Heaps, unlike contexts, are usually considered to be order-independent. We can express this order-independence by using the following language:

$$\begin{aligned} empty &: heap \\ bind &: (id!\!!, value) \to heap \\ merge &: (heap, heap) \to heap \end{aligned}$$

and specifying appropriate unit, commutativity, and associativity axioms

$$\begin{aligned} merge(empty, x) &= x = merge(x, empty) \\ merge(x, y) &= merge(y, x) \\ merge(x, merge(y, z)) &= merge(merge(x, y), z) \end{aligned}$$

This is quite similar to some formalisms for heaps employed in separation logic. In fact, separation logic is an important possible application for uniqueness types, as well as a rich source of interesting ideas. We are very interested in determining whether uniqueness types are subsumed by, or can be integrated with, related "resource-conscious" logics such as Bunched Implications [19] or Schöpp and Stark's dependent type theory for names and binding [24].

On the other hand, this technique for encoding finite partial maps only works for functions whose domain is a set of names. It would be interesting to see if there is a better way of dealing with finite partial maps on other types, such as lists of names (for namespaces) or pairs of state and alphabet symbols (for state transitions in automata, see the end of Section 5).

4.1 Namespaces

Open scoping techniques (such as global and sequential scoping) can be very awkward to use because of the need to manually avoid name collisions. For example, in a global scoping discipline, two programmers implementing separate modules must be careful not to duplicate identifiers. As a result, open scoping is often mediated by a hierarchical module, namespace, or interface system. We refer to all three situations using the generic term *namespaces*. Thus, an identifier only needs to be uniquely defined within its namespace. In addition, namespaces may be closed (that is, all components in the namespace are declared in a specific part of the program, as in ML modules) or open (that is, additional components can be added anywhere in the program, as in C++ namespaces).

There are two key problems relating to programming and formalizing namespace systems: first, how to faithfully encode languages with fully-qualified names, and second, how to resolve the partially- or unqualified names to fully-qualified names. Both seem interesting and difficult. Whether (and if so, how) namespaces can be supported using nominal logic (or any other abstract syntax formalism) is an important direction for future work.

5. Anonymous Data Structures

It is commonplace in some situations to think of all of the names in a data structure as being *anonymous* in the sense that they can all be renamed to fresh names without altering the meaning of the data structure. Three well-known examples include state names in a finite-state automata, variable names in a logic programming Horn clauses, and type variables in a ML polymorphic type scheme.

We propose a type constructor for *anonymous values* $\tau??$ for each type τ, and inhabited by terms of the form $t??$, such that

$$\frac{S \vdash t : \tau}{\varnothing \vdash t?? : \tau??}$$

Anonymous values are axiomatized as follows:

$$\frac{}{\mathsf{a} \# t??} \qquad \frac{((\mathsf{a}\ \mathsf{b}) \cdot t)?? \approx u??}{t?? \approx u??}$$

For finite terms, these axioms together imply that any anonymous value can be expressed as $t??$ for some t mentioning only fresh names.

For the Horn clause and ML type scheme examples, we could write

$$\begin{array}{rcl} tvar & : & var \to monotp. \\ tarr & : & (monotp, monotp) \to monotp. \\ polytp & : & monotp?? \to polytp. \\ \\ clause & : & (list\ goal \times atomic)?? \to clause. \end{array}$$

Of course, both examples can also be handled (and arguably better handled) by introducing an explicit universal quantifier for types or formulas and quantifying over all free variables. However, this technique damages adequacy: there is now more than one choice of representation for a formula/type of the form $T(\alpha, \beta) = f(\alpha, \beta) \to g(\alpha)$, namely $\forall\alpha.\forall\beta.T(\alpha, \beta)$ and $\forall\beta.\forall\alpha.T(\alpha, \beta)$. Of course, one way around this is to choose one "canonical" quantification ordering, such as the order in which each variable first occurs in the term (from left to right).

Another solution is to observe that semantically, $\forall\alpha.\forall\beta.T(\alpha, \beta)$ and $\forall\beta.\forall\alpha.T(\alpha, \beta)$ are equivalent; thus, the order of the quantifiers is irrelevant so it is safe to add axioms such as $\forall\alpha.\forall\beta.T \approx \forall\beta.\forall\alpha.T$ and $\alpha \# P \supset P \approx \forall\alpha.P$ that collapse the multiple possible representations. Then a formula or type is an equivalence class relating all α-renamings of representations of the form $\forall\vec{\alpha}.P$ where $FV(P) \subset \{\vec{\alpha}\}$. Such equivalence classes are in bijective correspondence with the inhabitants of anonymous types, since the latter are equivalence classes of objects up to renaming all free names. This seems essentially the same as using anonymous types, with the added complication of dealing with extraneous lists of quantifiers.

In addition, for automata, there is no obvious form of binding at hand: we really do (or at least, Hopcroft and Ullman really did [12]) consider automata to be equivalent up to permuting the state names. So, we can (to a first approximation) represent automata as an anonymous triple consisting of a start state, list of transitions, and list of final states.

$$aut : (Q \times list\ (Q \times \Sigma \times Q) \times list\ Q)?? \to nfa.$$

Indeed, Hopcroft and Ullman considered automata to be equivalent modulo an even richer equational theory: they represented the transitions using a function $\delta : \Sigma \times Q \to Q$ or relation $\Delta : Q \times \Sigma \times Q$, and represented the final states as a set. As a result, two structurally equivalent automata may not be equal as data structures. To repair this, we could add built-in set and finite map types, or equivalently add rules expressing the fact that automata are equal up to reordering the transition and final state lists. Another possibility would be to encode transition functions as functions $\Sigma \times Q \to Q$ and encode relations as functions $Q \times \Sigma \times Q \to bool$. While certainly adequate (and literally closer to what Hopcroft and Ullman had in mind), in a computationally rich theory (e.g. HOL or Coq) this encoding is more difficult to analyze because functions are black boxes.

6. Pattern Binding

We now consider a more complex situation in which we wish to bind all of (an unknown number of) names in a data structure; for example, to represent the syntax of ML-like pattern matching constructs:

```
case e of
  f(x,y) -> e1
| g(h([x,y]),z) -> e2
| ...
```

In the above expression, we view the occurrences of x and y in $f(x, y)$ as binding any occurrences in e_1, while x, y, z in $g(h([x, y]), z)$ are bound in e_2. Let us suppose for simplicity that patterns can consist of either a variable or a function symbol applied to a list of patterns. It is possible to encode the language of patterns as follows:

$$\begin{array}{rcl} pvar & : & id!! \to pat. \\ pfun & : & (fsym, list\ pat) \to pat. \\ match & : & blist \to match\_body. \\ case & : & (exp, list\ match\_body) \to exp. \\ bnil & : & (pat, exp) \to blist. \\ bcons & : & (\langle id \rangle blist) \to blist. \end{array}$$

(We use the uniqueness type to enforce the pattern variable linearity constraint). Note that the match case constructs a pattern-match from a pattern-expression pair, surrounded by a "binding list" that, intuitively, binds all of the names in the pattern (presumably in some canonical order, such as the order in which they occur in the pattern).

Then the above expression can be represented as

```
case(e,[match(bcons(<x>
              bcons(<y>
              bnil(pfun("f",[pvar(x),pvar(y)]),
                   e1)))),
        ...]).
```

We observe that this representation requires an intermediate processing step (calculating the sequence of free variables) to get from the original expression to the above. In addition, the representation type includes multiple distinct values that could correspond to the same input, obtained by reordering the variables in the $blist$. Thus, this encoding fails the "no junk" component of the adequacy property.

A more palatable alternative is to use standard syntax for patterns, such as

$$\begin{array}{rcl} pvar & : & id\text{!} \to pat. \\ pfun & : & (fsym, list\ pat) \to pat. \\ match & : & (pat, exp) \to match. \\ case & : & (exp, list\ match) \to exp. \end{array}$$

and then axiomatize the applicable renaming principles directly. This appears complicated: a name x that may be renamed in $p[x] \to e[x]$ may appear arbitrarily deeply in p. One possibility is to seek an appropriate generalization of α-equivalence for pattern-like bindings. Let $bnd(p, \mathsf{x})$ be a predicate testing whether pattern p binds name x, defined as

$$\frac{}{bnd(pvar(\mathsf{x}), \mathsf{x})} \quad \frac{bnd\_list(ps, \mathsf{x})}{bnd(pfun(\mathsf{f}, ps), \mathsf{x})}$$
$$\frac{bnd(p, \mathsf{x})}{bnd\_list(p :: ps, \mathsf{x})} \quad \frac{bnd\_list(ps, \mathsf{x})}{bnd\_list(p :: ps, \mathsf{x})}$$

Then the following rules for freshness and equality appear to suffice:[3]

$$\frac{bnd(p, \mathsf{x})}{\mathsf{x} \# match(p, e)}$$
$$\frac{bnd(p, \mathsf{x}) \quad \mathsf{x} \# p', e' \quad match(p, e) \approx (\mathsf{x}\ \mathsf{y}) \cdot match(p', e')}{match(p, e) \approx match(p', e')}$$

Thus, in addition to using ordinary properties of equality, we can show that two matches are equal provided we can find a names x and y bound in the first and second patterns respectively, such that x is not free in the scope of y and the matches are equal up to renaming x and y. For example, using a more readable syntax for matches, we can derive

$$\frac{(*) \quad \dfrac{(**) \quad \overline{(\mathsf{x}, \mathsf{y}) \to \mathsf{x} + \mathsf{y} + 1 \approx (\mathsf{x}, \mathsf{y}) \to \mathsf{x} + \mathsf{y} + 1}}{(\mathsf{x}, \mathsf{y}) \to \mathsf{x} + \mathsf{y} + 1 \approx (\mathsf{x}, \mathsf{z}) \to \mathsf{x} + \mathsf{z} + 1}}{(\mathsf{x}, \mathsf{y}) \to \mathsf{x} + \mathsf{y} + 1 \approx (\mathsf{y}, \mathsf{z}) \to \mathsf{y} + \mathsf{z} + 1}$$

(the omitted side conditions $(*)$ and $(**)$ are easily checked).

Also, recent work on the ρ-calculus [5] has considered extending the lambda-calculus with first-class pattern abstractions. The pattern abstraction $P \twoheadrightarrow M$ can be applied to any term N matching pattern P, and evaluating $(P \twoheadrightarrow M)\ N$ produces result $\sigma(M)$, provided $\sigma(P) = N$. The variables occurring in a pattern must be distinct, and the free variables of a pattern expression $P \twoheadrightarrow M$ are just $FV(M) - FV(P)$. Binding all of the distinct names of a term in another term seems to be a common enough case to deserve special attention and notation. Both FreshML [26] and Cαml [22] provide a similar feature.

[3] In the second rule, since $bnd(p, \mathsf{x})$ and $p \approx (\mathsf{x}\ \mathsf{y}) \cdot p'$, it follows that $bnd(p', \mathsf{y})$ also holds, so we do not need to check the latter.

7. Mutual Recursion with Pattern Matching

Mutual recursion is a convenient programming feature with quite complex binding structure. In this section we show how the techniques explored in the previous sections can be combined to axiomatize the binding behavior of mutual recursion.

We consider a mutual recursion construct similar to those of Standard ML, OCaml, Haskell, LISP, or Scheme. A general mutual recursive definition is written as follows:

$$\begin{array}{lllcl} letrec & f^1\ p_1^1 \cdots p_{n^1}^1 & = & e_1 \\ and & f^2\ p_1^2 \cdots p_{n^2}^2 & = & e_2 \\ \vdots & & & & \\ and & f^m\ p_1^m ... p_{n^m}^m & = & e^n \end{array}$$

Mutually recursive declarations exhibit the following structural behavior:

1. The names $f^1, \ldots, f^n$ must be distinct, and are all considered bound within each body $e_1, \ldots, e_n$ and in subsequent declarations.
2. All of the pattern variables of $p_1^k, \ldots, p_{n^k}^k$ must be distinct, and are considered bound in e_k.
3. The order of the function definitions is irrelevant.

The traditional way to handle `letrec` using unary lexical scoping is to replace it with a unary fixpoint operator such as $fix : \langle id \rangle exp \to exp$, and "de-sugar" simultaneous mutual recursive definitions to a single recursive definition of a n-tuple. This is completely satisfactory from a theoretical point of view, because the expressive power of `letrec` and of `fix` (in the presence of products) is exactly the same. From a pragmatic point of view, however, this encoding leaves something to be desired, partly because the de-sugaring translation from `letrec` to `fix` language itself requires care because of name-binding issues. Another possibility would be to use binding lists; however, this damages adequacy (as discussed in Section 6). We instead wish to consider whether the name-binding behavior of `letrec` can be axiomatized *as is*.

From a structural point of view, we can encode this syntax using the following definitions:

$$\begin{array}{rcl} fndecl & : & (fname\text{!}, list\ pat, exp) \to fndecl. \\ letrec & : & list\ fndecl \to decl. \end{array}$$

First, we assume that patterns are defined as in the last section, and that pattern variable uniqueness and binding are axiomatized for $fndecl(f, ps, e)$ as described in previous sections. We also enforce function name uniqueness using $(-)$!.

We apparently cannot axiomatize the renaming of the function names using sequential scoping axioms as outlined in Section 3. Were we to do so, the first function name would be considered bound in all the function bodies, as desired, but the second and subsequent names would not be considered bound in the first body. That is, we would have the binding structure of a nonrecursive `let`. In addition, care must be taken to ensure that the scope of the functions defined by the `letrec` extends to subsequent declarations in a declaration list: this requires looking "one level deeper" than was the case for the earlier sequential scoping examples.

We note that things do work fine for the first function declaration since all subsequent declarations follow it. If only we could consider *each* function declaration in parallel as "the first", then things would work out OK. In fact, the meaning of a `letrec` is usually independent of the order of function definitions, so we can safely identify `letrec` expressions up to reordering of the defini-

tions. This allows us to consider any of the simultaneous definitions to be "the first".

Therefore, we axiomatize function renaming for `letrec` as follows:

$$\frac{perm(ls, ls')}{letrec(ls) \approx letrec(ls')}$$

where $perm$ is a predicate expressing that two lists are permutations of one another; and

$$\frac{\begin{array}{c}\overline{\mathsf{f} \# letrec(fndecl(\mathsf{f}, ps, e) :: ls) :: ds} \\ \mathsf{f} \# (ps', e', ls', ds') \\ letrec(ps, e, ls) :: ds \approx (\mathsf{f}\ \mathsf{g}) \cdot (letrec(ps', e', ls') :: ds')\end{array}}{\begin{array}{c}letrec(fndecl(\mathsf{f}, ps, e) :: ls) :: ds \\ \approx letrec(fndecl(\mathsf{g}, ps', e') :: ls') :: ds'\end{array}}$$

that is, a function name f is fresh for a declaration list starting with a `letrec` that first defines f, and it is acceptable to rename the first `letrec`-function name f within its body, within the rest of the `letrec`, and in subsequent declarations.

It should be noted that this axiomatization is only acceptable when *all properties we care about* are preserved by reordering `letrec` cases. There are certainly situations (such as program transformation) in which it is not acceptable to reorder the cases: programmers do not want to use tools that make unnecessary syntactical changes. Consequently, we also propose an axiomatization that does not impose this additional structural congruence. In this axiomatization, we do not consider `letrec`s themselves equivalent up to reordering cases; instead, we define equality-up-to-renaming as a two-step process. First, we check to see whether a name is considered bound within the `letrec`, using a bnd predicate similar to that used for patterns; if so, we consider it to be fresh for (and renameable in) the whole `letrec` and subsequent names.

$$\frac{\overline{bnd((fndecl(\mathsf{f}, ps, e)) :: ls, \mathsf{f})}}{\overline{bnd((fndecl(\mathsf{g}, ps, e)) :: ls, \mathsf{f})}}$$

$$\frac{bnd(ls, \mathsf{f})}{\overline{\mathsf{f} \# letrec(ls) :: ds}}$$

$$\frac{\begin{array}{c}bnd(ls, \mathsf{f}) \quad \mathsf{f} \# (ls', ds') \\ letrec(ls) :: ds \approx (\mathsf{f}\ \mathsf{g}) \cdot (letrec(ls') :: ds')\end{array}}{letrec(ls) :: ds \approx letrec(ls') :: ds'}$$

8. Related and Future Work

The FreshML [25] and αProlog [2] programming languages provide unary lexical scoping as a language extension. However, there is no built-in support for more exotic forms of binding. More recently, Pottier has developed Cαml [22], a source-to-source translation tool that translates high-level binding specifications for OCaml types to low-level code that deals with names and binding automatically, using OCaml's object system. Interestingly, this approach is not limited to unary lexical scoping, and can even encode `letrec`. Another recent approach that provides some support for more general forms of binding is the *FreshLib* library [3], which implements much of the functionality of nominal abstract syntax as a Haskell class library. However, implementing custom binding forms such as pattern matching requires providing customized type class instances, so this approach is at present less declarative than Cαml's.

We have focused on expressiveness at the logical level without worrying about pragmatic issues such as the complexity of unification or typechecking needed for automation. While many of the examples are mild variants of alpha-equivalence and so can be handled efficiently using known techniques, other examples, especially those involving structural equivalences such as commutativity, associativity, and axioms like those of the π-calculus restriction operation, typically make unification at least **NP**-hard. We conjecture that most, if not all, interesting structural equivalences can be decided in polynomial time and unified in **NP**, and believe it will be interesting to seek out such well-behaved fragments of (and algorithms for) *nominal equational unification*. We also think it will be worthwhile to find more compact and declarative notations for the axioms for custom binding forms, since the approach we have used in this paper is verbose and error-prone.

While the uniqueness and anonymity types we have considered are interesting, they often do not quite express what we want. For example, we may wish to limit the anonymization performed by the anonymity type (or the binding of the names in a pattern) to a single name-type rather than all names present in a data structure.

Some of the scope situations we considered (in particular, hierarchical namespaces) appear to have no good solution so far. These problems require further study.

Although we have used nominal logic as a way of describing techniques for encoding the exotic binding forms we have considered, it is possible that other techniques are equally or more suitable. In particular, since a lot of prior research and implementation effort has focused on higher-order abstract syntax, it would be advantageous if techniques such as those we have discussed could be adapted to that setting. We do not see how to do this (which is one reason for our interest in nominal abstract syntax), but that does not mean it cannot be done.

9. Conclusion

In this paper we have identified some of the limitations of *unary lexical scoping*, the only form of scoping supported by most advanced techniques for representing languages with binding. We have described several common situations in which unary lexical scoping is either inconvenient or inadequate, and sketched how these forms of scoping can be formalized in nominal logic. In particular, we have shown that many structural equivalences can be axiomatized directly in nominal logic. We have also identified some potentially useful extensions to nominal abstract syntax, namely uniqueness types and anonymity types, which can be used to describe more exotic forms of binding.

Although we believe that this represents further evidence of the usefulness of nominal logic, these results are preliminary and there are several issues that need to be investigated in order for these techniques to be useful in real programming or reasoning systems. We wish to call attention to forms of binding, scoping, and structural congruence that appear "in the wild" (that is, in paper formalizations) and for which no abstract syntax encoding technique provides a satisfactory answer. Systematic techniques for encoding these syntactic constructs are important for bridging the gap between paper and machine-checked proofs of realistic programming language properties.

Acknowledgments

This paper was motivated partly by discussions with Andrew Pitts, Ian Stark, Peter Sewell, Matthew Fairbairn, and by recent discussions on the POPLMark mailing list. This work was supported by EPSRC grant R37476.

References

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000. `http://www.w3.org/TR/-2000/REC-xml-20001006`.

[2] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. 20th*

Int. Conf. on Logic Programming (ICLP 2004), number 3132 in LNCS, pages 269–283, 2004.

[3] James Cheney. Scrap your nameplate (functional pearl). In Benjamin Pierce, editor, *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 2005. To appear.

[4] James Cheney. A simpler proof theory for nominal logic. In *Proceedings of the 2005 Conference on Foundations of Software Science and Computation Structures (FOSSACS 2005)*, number 3441 in LNCS, pages 379–394. Springer-Verlag, 2005.

[5] Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Post-procededings of TYPES*, 2003.

[6] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, fifth edition, 2003.

[7] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[8] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Int. Conf. on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag LNCS 902.

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

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

[11] Furio Honsell, Marino Miculan, and Ivan Scagnetto. The theory of contexts for first order and higher order abstract syntax. In *TOSCA 2001 - Theory of Concurrency, Higher Order Languages and Types*, volume 62 of *Electronic Notes on Theoretical Computer Science*, 2001.

[12] John E. Hopcroft and Jeffrey D. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[13] Dale Miller. The pi-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1992.

[14] Dale Miller. Abstract syntax for variable binders: an overview. In John Lloyd et al., editor, *Computational Logic - CL 2000*, number 1861 in LNAI. Springer, 2000.

[15] Dale Miller. Encoding generic judgments: Preliminary results. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, volume 58(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[16] Dale Miller and Alwen Tiu. A proof theory for generic judgments: extended abstract. In *Proc. 18th Symp. on Logic in Computer Science (LICS 2003)*, pages 118–127. IEEE Press, 2003.

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[18] A. Momigliano and Simon Ambler. Multi-level meta-reasoning with higher order abstract syntax. In *FOSSACS 2003*, pages 375–391. Springer-Verlag, 2003.

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

[20] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89)*, pages 199–208. ACM Press, 1989.

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

[22] François Pottier. An overview of Cαml, June 2005. Available at `http://cristal.inria.fr/~fpottier/publis/-fpottier-alphacaml.pdf`.

[23] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[24] Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In *Proceedings of the 2004 Computer Science Logic Conference*, number 3210 in Lecture notes in Computer Science, pages 235–249, Karpacz, Poland, 2004.

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

[26] M.R. Shinwell and A.M. Pitts. Fresh objective caml user manual. Technical Report 621, Cambridge University Computer Laboratory, February 2005.

[27] Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *Proceedings of the 3rd EATCS Workshop in the Foundations of Global Computing (FGUC 2004)*, 2004. To appear in ENTCS.

[28] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

[29] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proceedings of the 2000 European Symposium on Programming*, number 1782 in LNCS. Springer-Verlag, 2000.