

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the [ENTCS Macro Home Page](#).

Representing names with variables in nominal abstract syntax

Matthew Lakin^{1,2}

*University of Cambridge
Computer Laboratory
Cambridge, UK*

Abstract

We describe a new approach to nominal abstract syntax where object-language names are represented using *variables* in the meta-language, as opposed to concrete atoms. As well as providing additional flexibility through the ability to perform *aliasing* between bound and free names, this approach more closely models informal practice in the specification of inductively-defined relations. We present a meta-language called α ML which includes these features and develop a theory of contextual equivalence for this language. Our main result is that the ability to generate fresh atoms (as in FreshML and α Prolog) is *not* required to achieve an embedding of syntax trees involving binders for which α -equivalence of trees corresponds to contextual equivalence in the meta-language.

Keywords: Binders, alpha-equivalence, nominal abstract syntax, meta-programming, functional logic programming, operational semantics, contextual equivalence.

1 Introduction

Traditional nominal abstract syntax (NAS) represents abstract syntax in terms of *ground trees*, g :

$$g ::= a \mid K g \mid (g, \dots, g) \mid \langle a \rangle g$$

Existing nominal meta-programming languages such as FreshML [9] and α Prolog [1] follow this approach, representing object-level names using *atoms* a . The only operations permitted on atoms are equality testing and the generation of a *fresh* atom that has not been seen before. An object-level (single) binder is modelled by the *abstraction* term-former $\langle a \rangle (-)$ of nominal logic [6], where the name modelled by the atom a is bound in the body.

The syntax of these languages forces the user to write an atom as the first argument to the abstraction term-former, i.e. if x is a variable then a tree such as $\langle x \rangle (-)$ is not grammatical. Furthermore, the abstraction term-former itself is

¹ Work supported by UK EPSRC grant EP/D000459/1

² Email: Matthew.Lakin@cl.cam.ac.uk

not a binder in the meta-language. This means that if $a \neq a'$ are two distinct atoms then the ground trees $\langle a \rangle a$ and $\langle a' \rangle a'$ are *not* the same. However, there is a well-defined notion of α -equivalence $=_\alpha$ on such trees, and clearly we should have $\langle a \rangle a =_\alpha \langle a' \rangle a'$. This raises the issue of how to represent abstractions using atoms such that α -equivalence is respected. The approach taken for FreshML in [9,8] is to prove that two encoded terms behave identically in the operational semantics *precisely when* they are α -equivalent. We adopt a modified version of this approach.

If concrete atoms may appear in programs then programs exist which violate the fundamental *equivariance* assumption of nominal logic, which states that “*validity is invariant under swapping names.*” α Prolog is incomplete for nominal logic precisely because it fails to account for equivariance [1,10]. Representing object-level names using *variables* and doing without atoms altogether would give us equivariance for free, as there would be no atoms to swap.

An alternative approach is to use higher-order abstract syntax (HOAS) [5] where object-level binding is modelled using meta-level binding. Atoms are not used but the bound variables are anonymized, therefore the programmer cannot get an explicit handle on the binders. This can make it difficult to translate some pen-and-paper definitions into a HOAS form, in particular those which rely on *aliasing* between names. An example is the `spec` relation defined by the following two rules

$$\frac{}{\text{spec}(\text{mono}(T), [], T)} \qquad \frac{\text{spec}(T, \vec{x}, T')}{\text{spec}(\text{all} \langle x \rangle T, x :: \vec{x}, T')}$$

which unbinds the bound variables of a polymorphic type scheme (the first argument), producing a list of the unbound variables and the underlying type scheme. Incompleteness is an issue in the second rule because aliasing is needed between the bound variable in the type scheme and the free occurrence in the list.

We argue that neither HOAS nor NAS with atoms *quite* models informal practice. Mathematicians typically use variables in binding position to range over *any* bound name in the object-language but also need direct access to that bound name for certain definitions to fall out elegantly. This leads us to develop a meta-language, α ML, which deals with binders using nominal abstract syntax yet refers to object-level names using *variables* in the same way as any other object-level term. This affords additional flexibility without sacrificing any of the functionality of existing nominal meta-programming languages.

2 α ML

The intended application of α ML is as a target language to produce executable prototypes from a semantic description of a language. Its syntax combines features of functional and logic programming, and was specifically designed to provide a seamless embedding of inductive definitions (see [3] for more details). The expressions of the language are defined by the following grammar. We assume that variables x ,

f etc. are implicitly typed and are drawn from some countably infinite set.

$$e ::= x \mid K e \mid (e, \dots, e) \mid \text{fun } f x = e \mid \underline{\top} \mid \langle x \rangle e \mid \text{let } x = e \text{ in } e \mid e e \mid \\ e.i \mid \text{case } e \text{ of } K x \Rightarrow e \mid \dots \mid K x \Rightarrow e \mid \underline{\exists x}. e \mid \underline{e \text{ or } e} \mid \underline{e = e} \mid x \# e$$

The binding constructs are as usual, recalling that the abstraction term-former $\langle x \rangle -$ is *not* a meta-level binder. For brevity we only describe the non-standard constructs, which are underlined above (also the syntax $e.i$, which projects the i^{th} element of a tuple). αML has a straightforward, monomorphic type system which we do not describe here for reasons of space, except to note that it delineates a class of special *equality types*. These have a decidable notion of equality and correspond to the ground trees from the Introduction (with atoms replaced by variables x).

The expression $\langle x \rangle e$ represents the object-level binding of the name represented by the variable x in its lexical scope—the value produced by evaluating e . The type system ensures that e has an equality type (the FreshML type system is less restrictive, allowing the body of an abstraction to have a higher type, for example) so the abstraction terms correspond to binders in the object-language. The use of variables in binders means that an abstraction value $\langle x \rangle \langle x' \rangle x$ denotes a *set* of possible instantiations including $\langle a \rangle \langle b \rangle a$ and $\langle a \rangle \langle a \rangle a$, which are *not* α -equivalent.

The value \top is the result of a successful proof-search computation. The *constraints* $v = v'$ and $x \# v$ are (α -)equality and *freshness* respectively. Freshness coincides with the usual notion of “not a free variable of” (the α -equivalence class of) an object-level term. The expression $\exists x. e$ dynamically generates a *new* variable of some equality type, which could represent an object-level name or data term, depending on the type of x . Finally, the e_1 or e_2 construct is a non-deterministic branch which is used to model proof-search computations.

The *operational semantics* is defined by a non-deterministic small-step transition relation \longrightarrow between configurations of the form $\exists \vec{x}(\bar{c}; F; e)$, which record the generated variables of equality type (\vec{x}), the accumulated constraint set (\bar{c}) and a continuation (F) expressed as a *frame stack* [7]. The transition rules for the \longrightarrow relation include the usual context reduction rules for a call-by-value functional programming language, and rules for implementing “flexible” case analysis and projection with narrowing, similar to Curry [2]. The following rules define transitions for the novel constructs of αML .

- $\exists \vec{x}(\bar{c}; F; c) \longrightarrow \exists \vec{x}(\bar{c} \ \& \ c; F; \top)$ if $\models \exists \vec{x}(\bar{c} \ \& \ c)$.
- $\exists \vec{x}(\bar{c}; F; \exists x. e) \longrightarrow \exists \vec{x}, x(\bar{c}; F; e)$ if $x \notin \vec{x}$ and $\models \exists x(\top)$.
- $\exists \vec{x}(\bar{c}; F; e_1 \text{ or } e_2) \longrightarrow \exists \vec{x}(\bar{c}; F; e_i)$ if $i \in \{1, 2\}$.

The first rule above can only fire if the new constraint c is satisfiable in conjunction with the existing constraints (checking this is NP-complete [3]). Otherwise the configuration is stuck and there is no specified behaviour—we do not prescribe any particular backtracking strategy. The second rule produces a variable that does not already appear in \vec{x} by exploiting α -conversion in the meta-language, and the $\models \exists x(\top)$ side-condition means that there does exist some finite tree with the same type as x . The final rule introduces non-determinism by picking to execute either e_1 or e_2 . Again, the search strategy is unspecified in the operational semantics.

3 Contextual equivalence for α ML

We present operational equivalence of α ML expressions in terms of observing *termination* only. A configuration *may succeed*, written $\exists \vec{x}(\vec{c}; \mathbf{F}; e)\downarrow$, if some finite sequence of \longrightarrow transitions leads to a configuration of the form $\exists \vec{x}(\vec{c}; \text{Id}; v)$, where Id is the identity continuation and $\models \exists \vec{x}(\vec{c})$ holds. We define a relation of operational equivalence $\vec{x} \vdash e \cong e' : \mathbf{T}$ which holds iff e and e' have free variables in \vec{x} (of equality types), both have the same type and for any (well-typed) contexts $\exists \vec{x}(\vec{c}; F; -)$, it is the case that

$$\exists \vec{x}(\vec{c}; F; e)\downarrow \iff \exists \vec{x}(\vec{c}; F; e')\downarrow.$$

We generalise to a relation \cong° between arbitrary expressions by quantifying over all closing substitutions. The following result is after [4], and enumerates certain desirable properties of \cong° and shows that it coincides with contextual equivalence.

Theorem 3.1 (CIU) *Operational equivalence (\cong°) is an equivalence relation and is adequate for observing termination. It is preserved by substitution and all term-formers of the α ML language, and is also the largest such expression relation. \square*

4 Representing α -equivalence classes

Pitts and Shinwell [8] prove a “correctness of representation” result for FreshML, that α -equivalence and contextual equivalence coincide for expressions which represent ground trees. A similar result holds for α ML. This is not obvious because α ML includes neither atoms nor a **fresh** operator for generating fresh atoms—this complicates the encoding of ground trees g into α ML. We first fix a sort-preserving bijection between every atom \mathbf{a} and the corresponding variable $\mathbb{V}_{\mathbf{a}}$ (for translating free atoms) and introduce a notion of sort-preserving *environments* E which map finitely many atoms to variables (for dealing with abstracted atoms). Then, writing $\text{fa}(g)$ for the *free atoms* of g , we define its translation $\llbracket g \rrbracket$ as

$$\llbracket g \rrbracket \triangleq \exists(\vec{x}' - \vec{x}_g). \#_{\vec{x}'} \& v_g$$

where $\#_{\vec{x}'}$ abbreviates the conjunction of all freshness (i.e. inequality) constraints $x_i \# x_j$ for $x_i, x_j \in \vec{x}'$, where $i \neq j$. The $\&$ operator is a defined syntactic sugar that models sequential composition using the eager semantics of α ML. The list of variables \vec{x}' and the value v_g are derived using the \triangleright relation defined below, where $E_g \vdash \langle \vec{x}_g; g \rangle \triangleright \langle \vec{x}'; v_g \rangle$ holds. For all and only $\mathbf{a} \in \text{fa}(g)$, (i) the environment E_g maps \mathbf{a} to its translation $\mathbb{V}_{\mathbf{a}}$ under the fixed bijection, and (ii) the variable $E_g(\mathbf{a})$ appears in \vec{x}_g .

$$\frac{E(\mathbf{a}) = x}{E \vdash \langle \vec{x}; \mathbf{a} \rangle \triangleright \langle \vec{x}; x \rangle} \qquad \frac{x \notin \vec{x} \quad E[\mathbf{a} \mapsto x] \vdash \langle x :: \vec{x}; g \rangle \triangleright \langle \vec{x}'; v \rangle}{E \vdash \langle \vec{x}; \langle \mathbf{a} \rangle g \rangle \triangleright \langle \vec{x}'; \langle x \rangle v \rangle}$$

$$\frac{E \vdash \langle \vec{x}; g \rangle \triangleright \langle \vec{x}'; v \rangle}{E \vdash \langle \vec{x}; K g \rangle \triangleright \langle \vec{x}'; K v \rangle} \qquad \frac{E \vdash \langle \vec{x}; g_1 \rangle \triangleright \langle \vec{x}_1; v_1 \rangle \quad \dots \quad E \vdash \langle \vec{x}_{k-1}; v_{k-1} \rangle \triangleright \langle \vec{x}'; v_k \rangle}{E \vdash \langle \vec{x}; (g_1, \dots, g_k) \rangle \triangleright \langle \vec{x}'; (v_1, \dots, v_k) \rangle}$$

The judgement $E \vdash \langle \vec{x}; g \rangle \triangleright \langle \vec{x}'; v_g \rangle$ relates g to a value v_g via the environment E and the lists of variables \vec{x} and \vec{x}' . The rule for tuples demonstrates how the

lists of variables are threaded through, recording the variables used to implement bound names. The atom a becomes the variable $E(a)$ using the environment. The FreshML translation of an abstraction uses the `fresh` operator to produce fresh atoms to stand for bound variables—the α ML version uses \exists -bound variables to model bound names (chosen to avoid \vec{x} , which prevents clashes). We then mimic the behaviour of `fresh` by manually asserting that the bound and free variables are all pairwise distinct. In this rule the environment is updated by mapping a to the newly-chosen variable x , which overrides any existing mapping for a . Thus environments implement the lexical scope of bound names in the natural way.

Theorem 4.1 (Fundamental correctness property) *If the variables \vec{x} correspond to $\text{fa}(g, g')$ under the fixed bijection then $g =_{\alpha} g' : E$ iff $\vec{x} \vdash \llbracket g \rrbracket \cong \llbracket g' \rrbracket : E$.*

Proof. The forward direction is straightforward because the α ML translations of α -equivalent ground trees differ only by α -renaming the \exists -bound variables, and the result follows because \cong is reflexive. The reverse direction is more complicated and involves intermediate lemmas, including an induction over the \triangleright relation. \square

5 Conclusion

We have demonstrated that bindable names may be represented using variables and freshness constraints, in the absence of concrete atoms and fresh atom generation which feature in FreshML and α Prolog. This representation is provably correct in that ground trees are α -equivalent iff their encodings in α ML are contextually equivalent. The fact that atoms do not appear in α ML programs means that equivariance is just an interesting meta-theoretic result about the semantics, rather than a property of programs which must hold for that particular program to be meaningful.

References

- [1] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. ICLP 2004*, volume 3132 of *LNCS*, pages 269–283. Springer-Verlag, 2004.
- [2] M. Hanus. Multi-paradigm declarative languages. In *Proc. ICLP 2007*, volume 4670 of *LNCS*, pages 45–75. Springer-Verlag, 2007.
- [3] M. R. Lakin and A. M. Pitts. Resolving inductive definitions with binders in higher-order typed functional programming. In *Proc. ESOP 2009*, volume 5502 of *LNCS*, pages 47–61. Springer-Verlag, 2009.
- [4] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
- [5] D. A. Miller. Abstract syntax for variable binders: An overview. In *CL 2000 Proceedings*, volume 1861 of *LNAI*, pages 239–253. Springer-Verlag, 2000.
- [6] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. and Comput.*, 186:165–193, 2003.
- [7] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.
- [8] A. M. Pitts and M. R. Shinwell. Generative unbinding of names. *Logical Methods in Computer Science*, 4(1:4):1–33, 2008.
- [9] M. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge, 2005.
- [10] C. Urban and J. Cheney. Avoiding equivariance in Alpha-Prolog. In *Proc. TLCA 2005*, volume 3461 of *LNCS*, pages 74–89. Springer-Verlag, 2005.