Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Encoding abstract syntax without fresh names

Matthew R. Lakin · Andrew M. Pitts

Received: date / Accepted: date

Abstract This paper introduces a variant of nominal abstract syntax in which bindable names are represented by normal meta-variables as opposed to a separate class of globally fresh names. Distinct meta-variables can be instantiated with the same concrete name, which we call aliasing. The possible aliasing patterns are controlled by explicit constraints on the distinctness (freshness) of names. This approach has already been used in the nominal meta-programming language αML. We recap that language and develop a theory of contextual equivalence for it. The central result of the paper is that abstract syntax trees (ASTs) involving binders can be encoded into αML in such a way that α-equivalence of ASTs corresponds with contextual equivalence of their encodings. This is novel because the encoding does not rely on the existence of globally fresh names and fresh name generation, which are fundamental to the correctness of the pre-existing encoding of abstract syntax into FreshML.

Keywords Meta-programming · Alpha-equivalence · Nominal abstract syntax

1 Introduction

This paper is about the representation of names in nominal abstract syntax (NAS) [12, 10]. NAS aims to model informal mathematical practice, where it is natural to write down bound names using some explicit symbol. For example, theoreticians would tend to write down the K combinator of the λ-calculus [1] in an explicitly named form such as $\lambda x.\, \lambda y.\, x$ as opposed to a nameless de Bruijn representation [2] such as $\lambda\, \lambda\, 2$.

Research supported by UK EPSRC grant EP/D000459/1.

Matthew R. Lakin
University of Cambridge Computer Laboratory, Cambridge, UK
E-mail: Matthew.Lakin@cl.cam.ac.uk
Present address: Microsoft Research, Cambridge, UK

Andrew M. Pitts
University of Cambridge Computer Laboratory, Cambridge, UK
E-mail: Andrew.Pitts@cl.cam.ac.uk

Object-level names are typically represented as elements drawn from some countably infinite set, which we will call *Name*. We refer to them indirectly via the meta-variable n. It is common practice to assume that these meta-variables range permutatively over *Name*, i.e. that syntactically distinct meta-variables n_1 and n_2 always refer to distinct names. This is referred to by Gabbay as the *permutative convention* [11]. We feel it is important to highlight these assumptions which are taken for granted when dealing with binders in an explicitly named form, as we will remove them later in the paper. Recalling the K combinator example above, many would automatically assume that x and y represent distinct names.

In this paper we will discuss a generalisation of traditional nominal techniques which we refer to as *non-permutative* nominal abstract syntax (NPNAS), with particular emphasis on its interplay with program equivalence. To motivate this, consider the case of instantiating first-order schematic patterns which involve meta-variables x, y etc. In first-order abstract syntax, syntactically different meta-variables can be instantiated with the same value. For example, the schematic pattern $(x\,,y)$ could be instantiated to $(2\,,3)$ using the substitution $[x \mapsto 2, y \mapsto 3]$ but it could also be instantiated to $(3\,,3)$ by the substitution $[x \mapsto 3, y \mapsto 3]$. We refer to this phenomenon as *aliasing* and say that x and y are *aliased* here. Note, however, that the pattern $(x\,,x)$ cannot be instantiated to $(2\,,3)$ because all occurrences of x must be instantiated consistently.

Now consider the case where the schematic patterns may involve bindable names. In the traditional NAS approach, bindable names are represented using a different class of meta-variables (n here) which range permutatively over *Name*. In NPNAS we remove the permutative convention on bindable names and represent them using normal meta-variables x, y etc. which may be aliased. Instead of the permutative convention we maintain explicit distinctness information about names in a separate constraint environment. NPNAS has greater expressive power than traditional NAS because aliasing is possible. This means that each schematic pattern may denote a set of different ASTs, depending on the topology of aliasing between its bound names. However, we can simulate permutative behaviour over a known finite set of names by adding explicit distinctness constraints between those names. For example, the schematic NPNAS equivalent of the K combinator mentioned above is

$$\texttt{Lam <}x\texttt{>(Lam <}y\texttt{>(Var }x\texttt{))} \tag{1}$$

where we use a standard nominal datatype declaration for λ-terms (introduced in Section 2.1 below). However, this could represent either or both of the distinct α-equivalence classes

$$(i)\quad [\texttt{Lam <}n\texttt{>(Lam <}n'\texttt{>(Var }n\texttt{))}]_\alpha \qquad\qquad (ii)\quad [\texttt{Lam <}n\texttt{>(Lam <}n\texttt{>(Var }n\texttt{))}]_\alpha$$

depending on whether we assert that $x \neq y$ (just (i)), $x = y$ (just (ii)), or leave it unspecified (both (i) and (ii)). We recall that the names n and n' follow the permutative convention, so we know that $n \neq n'$ and hence the intended meaning of (1) is the α-equivalence class (i).

We have practical experience of the NPNAS approach in the context of the functional logic programming language αML [19]. This is a call-by-value, higher-order, typed meta-programming language. It includes features for implementing α-equivalence-respecting inductive definitions involving binders, such as the ability to create and deconstruct name abstractions, support for generating names and for aliasing, constraints

of equality and name freshness, and fair proof-search by non-deterministic branching. The intended application of αML is as an executable meta-language for compiling a semantic description of an object-language into a prototype. We will present a brief overview of the language in Section 2.

Remark 1 (Practical motivation for NPNAS) We developed the NPNAS technique for use in αML to overcome incompleteness issues with proof-search. This situation is similar to that encountered in the nominal logic programming language αProlog [6, Section 5.3] if the tractable nominal unification algorithm [34] is used for matching against clauses. In this case, there are some programs for which all solutions cannot be found because the system does not take equivariance into account (such as [7, Example 5.17]). Various solutions have been proposed, such as imposing syntactic restrictions on αProlog programs [33,7] or moving to the more powerful *equivariant unification* algorithm [5]. Neither is ideal: in particular, the equivariant unification is rather involved and complicated to implement.

By abandoning the permutative convention altogether in αML we obtain a simpler language and constraint problem. Both this constraint problem and the equivariant unification problem are known to be NP-complete (see [3] and [19, Section 3] for proofs). Thus they are both polynomial-time reducible to each other and hence our constraint problem is equivalent to equivariant unification. We showed in [19] that proof-search in αML is complete for a simple yet powerful model of inductive definitions involving binders. Hence, the motivation for adopting NPNAS comes from the (constraint) logic programming side of αML. In this paper we focus on program equivalence which is largely concerned with the functional programming aspects of the language, so we will not discuss logic programming in great detail.

In [19] we claimed that αML correctly implements names and binding modulo α-equivalence. To back up this claim we must show a correctness result for αML akin to the following theorem for the FreshML language, which was proved in [30] and [27].

Theorem 1 (Correctness of representation for FreshML.) *Two λ-terms are α-equivalent, $t_1 =_\alpha t_2$, iff their representations $[t_1]_F$ and $[t_2]_F$ are contextually equivalent clased values of type* `term`*, i.e. can be used interchangeably in any well-typed Fresh Objective Caml program without affecting the observable results of program execution.* □

FreshML is a prime example of the nominal approach. It extends a higher-order typed functional programming language with convenient constructs for encoding names and binders, where names n are assumed to follow the permutative convention. Names are introduced by *fresh name generation*. Evaluating the `fresh` keyword returns a name which has never been seen before and hence may be assumed to be distinct from all other names in the evaluation context. Abstractions are created using a term-former which takes a name n and a value v and produces a value $\texttt{<}n\texttt{>}v$ denoting the object-level binding of the name n in v. An abstraction $\texttt{<}n\texttt{>}v$ in FreshML can only be deconstructed by *generative unbinding* [27], which generates another fresh name n' and returns a freshened copy $v[n'/n]$ of the abstraction body, along with n' itself. The analogue of Theorem 1 for the αML meta-language is by no means obvious, because the αML does not assume the permutative convention and does not provide a facility for fresh name generation, which is a crucial part of the encoding of abstract syntax into FreshML. Therefore, we will need a different encoding of abstract syntax in terms of the facilities available in αML.

The rest of the paper is organised as follows. In Section 2 we give a brief overview of the αML meta-language. The technical contribution of the paper begins in Section 3, where we present a formal notion of abstract syntax trees with binders and describe an encoding of these into αML. In Section 4 we develop a theory of contextual equivalence between αML expressions and in Section 5 we prove the central result of the paper concerning the representation of ASTs with binders in αML. We highlight related and future work in Section 6 and conclude in Section 7.

2 The αML language

In this section we give a brief overview of the syntax, type system and operational semantics of the αML meta-language, which was introduced in [19].

2.1 Types

Object-language abstract syntax is modelled using recursive datatype definitions. These extend the algebraic datatypes of functional programming languages such as Standard ML [22] to carry information about name-binding. A datatype declaration Σ in αML consists of

- a finite set $\mathbb{N}_\Sigma$ of *name sorts* N;
- a finite set $\mathbb{D}_\Sigma$ of *data sorts* D (disjoint from $\mathbb{N}_\Sigma$); and
- a finite set $\mathbb{C}_\Sigma$ of *constructors* $K\mathtt{:}T \to D$, where the argument types $T \in \mathit{Ty}_\Sigma$ are generated by the following grammar.

$$
\begin{array}{rll}
T \in \mathit{Ty}_\Sigma ::= & E & \text{(equality types)} \\
& D & \text{(data sorts)} \\
& T * \cdots * T & \text{(product types)} \\
& T \to T & \text{(function types)} \\
& \mathtt{prop} & \text{(type of semi-decidable propositions).}
\end{array}
$$

Equality types, $E \in \mathit{Ety}_\Sigma$, are a subset of αML types which have a decidable notion of equality between their inhabitants. These are generated by the following grammar.

$$
\begin{array}{rll}
E \in \mathit{Ety}_\Sigma ::= & S & \text{(nominal data sorts)} \\
& N & \text{(name sorts)} \\
& \mathtt{[}N\mathtt{]}\,E & \text{(abstraction types)} \\
& E * \cdots * E & \text{(product types)} \\
& \mathtt{unit} & \text{(unit type).}
\end{array}
$$

The meta-variable S ranges over *nominal* data sorts, which are the subset $\mathbb{S}_\Sigma \subseteq \mathbb{D}_\Sigma$ such that if $S \in \mathbb{S}_\Sigma$ then all constructors for S have types $K\mathtt{:}E \to S$, i.e. they take equality types as arguments. This stratification is necessary statically to ensure that we only encounter decidable constraints between values during the evaluation of αML programs. In particular, constraints between higher-order values would be undecidable [13], so $E \to E$ is not an equality type.

The novel types are the name sorts, abstraction types and the `prop` type. The only values of name sorts N are meta-variables which denote object-level bindable names of that sort. The abstraction type $\mathtt{[}N\mathtt{]}\,E$ represents the object-level binding of

Values, v	::=	x, f	(variable)
		$K\,v$	(constructor application)
		`()`	(unit)
		$(v, \ldots, v)$	(tuple)
		`fun` f `(`x`:`T`)`:T `=` e	(recursive function)
		`T`	(success)
		`<`v`>`v	(name abstraction)
Constraints, c	::=	v `=` v	(equality constraint)
		v `#` v	(freshness constraint)
Expressions, e	::=	v	(value)
		`let` x `=` e `in` e	(`let` binding)
		$v\,v$	(function application)
		`case` v `of` $K\,x$ `->` e `|` $\cdots$ `|` $K\,x$ `->` e	(`case` expression)
		$v.\,i$	(projection)
		c	(constraint)
		$\exists x{:}E.\,e$	(existential)
		$e \parallel e$	(non-deterministic branch)
Frame stacks, F	::=	`Id`	(empty stack)
		$F \circ (x.\,e)$	(non-empty stack).

Fig. 1 αML syntax

a (single) name of sort N in a term of type E. Note that the body of an abstraction must always be an equality type—this is necessary for constraints between abstractions to be decidable. This restriction does not apply in FreshML because abstractions are deconstructed using name-swapping rather than constraint solving. Finally, expressions of type `prop` correspond to proof-search and constraint solving computations from the world of constraint logic programming (CLP) [17]. We call these semi-decidable because the denotations of these relations are not recursively enumerable in the general case.

A pleasing consequence of this design is that the *nominal signatures* of [34] correspond to the subset of αML datatype declarations in which all data sorts are actually nominal data sorts. For example, the nominal signature for untyped λ-terms corresponds to an αML datatype definition which contains a single name sort `var` and a single nominal data sort `term` and has three constructors, whose types are as follows.

`Var : var` $\rightarrow$ `term`	(variables)
`App : term * term` $\rightarrow$ `term`	(applications)
`Lam : [var]term` $\rightarrow$ `term`	(λ-abstractions).

Henceforth we use this datatype declaration to encode λ-terms into αML.

2.2 Syntax

Fix a countably infinite set *Var* of variables, ranged over by x, which do *not* follow the permutative convention. These will represent unknown meta-level values, which include unknown object-level terms and bindable names. The syntax of αML is presented in Figure 1. The grammars of values and expressions correspond to a higher-order functional programming language extended with additional features for producing executable prototypes of inductively-defined relations. We only consider expressions which are in A-normal form [9], which means that evaluation order is specified using

`let` bindings. This simplifies the presentation and makes proofs more straightforward, without reducing the expressiveness of the language (extra `let` bindings can be added to translate more general constructs such as $\texttt{<}e\texttt{>}e'$ into A-normal form). We identify expressions up to α-conversion of bound variables. The variables in recursive functions, `let` bindings and `case` expressions bind as one would expect—the only unusual binding form is $\exists x\!:\!E.\, e$, where x is bound within e.

In the interest of brevity we will not discuss the syntactic constructs which are present in a standard functional programming language. Turning to the novel syntactic constructs, the value T represents successful completion of some CLP-style computation. The abstraction term-former $\texttt{<}v\texttt{>}v'$ represents an object-level name-binder, with the single name v bound in its lexical scope v'. The typing relation presented in Figure 2 will ensure that v is always some variable x of name sort. We stress that the abstraction term-former is *not* a binder in the meta-language. This means that we regard $\texttt{<}x\texttt{>}x$ and $\texttt{<}y\texttt{>}y$ as distinct expressions when $x \neq y$.

Constraints can either be equality or freshness constraints. Equality constraints correspond to α-equivalence of ASTs and freshness constraints model the "not free in" relation between a name and an AST. This is a common side-condition in definitions which is used to prevent name capture. The type system presented below ensures that v in the freshness constraint $v \,\#\, v'$ is always some variable x of name sort. In the case where v' is also some variable x' of name sort, the constraint $x \,\#\, x'$ corresponds to a *name inequality* test between x and x'. Practical experience has shown that this is a concise yet expressive constraint grammar. We write $\overline{c}$ for a finite conjunction of atomic constraints $c_1 \,\&\, \cdots \,\&\, c_n$ and identify the empty conjunction with the true expression T. It was noted in [19] that satisfiability of these constraint problems is NP-complete. The details of constraint solving are not relevant to this paper so we will not discuss them further.

The expression $\exists x\!:\!E.\, e$ uses meta-level α-renaming to generate a new meta-variable of equality type E, which is bound in the expression e. In the case when E is a name sort N, the variable denotes an unknown object-level bindable name. Although the meta-variables which we generate are always new in the meta-level, they do not necessarily denote fresh names at the object-level because we have abandoned the permutative convention. For example, the three possible instantiations of the schematic version of the K combinator from (1) can all be rendered in αML by the judicious addition (or omission) of freshness constraints. The language also includes a non-deterministic branching operator $e_1 \parallel e_2$ for modelling proof-search computations.

We use evaluation contexts in the style of Felleisen [8], formalised using frame stacks F after [24]. These are also defined in Figure 1. A frame stack is the continuation which corresponds to the remainder of the current computation. The empty stack `Id` means that we have nothing else to do with the result from the expression currently being evaluated, and the non-empty stack $F \circ (x.\, e)$ binds the result v of the current expression to x in e and evaluates this in the context F (where x is bound in e and we identify frame stacks up to α-conversion).

2.3 Type system

We now turn to the αML type system. We distinguish three kinds of type environment: Γ ranges over finite partial functions from Var to Ty_Σ, Δ over finite partial functions from Var to Ety_Σ and η over finite partial functions from Var to $\mathbb{N}_\Sigma$. We write $\mathit{dom}(\Gamma)$

$$\frac{}{\Gamma \vdash \mathtt{T} : \mathtt{prop}} \qquad \frac{\Gamma \vdash v : N \qquad \Gamma \vdash v' : E}{\Gamma \vdash \mathtt{<}v\mathtt{>}v' : \mathtt{[}N\mathtt{]}\,E} \qquad \frac{\Gamma \vdash v : E \qquad \Gamma \vdash v' : E}{\Gamma \vdash v \mathtt{=} v' : \mathtt{prop}}$$

$$\frac{\Gamma \vdash v : N \qquad \Gamma \vdash v' : E}{\Gamma \vdash v \mathtt{\#} v' : \mathtt{prop}} \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash e' : T}{\Gamma \vdash e \parallel e' : T}$$

$$\frac{x \notin dom(\Gamma) \qquad \alpha\text{-}Tree_{\Sigma}(E) \neq \emptyset \qquad \Gamma, x : E \vdash e : T}{\Gamma \vdash \exists x : E.\, e : T} \qquad \frac{}{\Gamma \vdash \mathtt{Id} : T \to T}$$

$$\frac{\Gamma \vdash F : T'' \to T' \qquad \Gamma, x : T \vdash e : T'' \qquad x \notin dom(\Gamma)}{\Gamma \vdash F \circ (x.\, e) : T \to T'}$$

Fig. 2 Selected αML typing rules

for the domain of definition of Γ (similarly for Δ and η). The typing judgement for expressions has the form $\Gamma \vdash e : T$, and typing rules for the novel forms of expression are presented in Figure 2. The rules corresponding to standard features of functional programming languages are entirely standard. We force programmers to provide explicit types for recursive function values and existentially-quantified variables—this reduces the typeability problem to type checking and forces users to document their code in the form of type annotations.

The rule for an abstraction $\mathtt{<}v\mathtt{>}v'$ requires that the value v in abstraction position is of some name sort N. By inspection of the rules we see that this only possible if v is a variable x such that $x \in dom(\Gamma)$ and $\Gamma(x) = N$ (because the sets $\mathbb{D}_{\Sigma}$ and $\mathbb{N}_{\Sigma}$ are disjoint). We have already discussed the requirement that the body of an abstraction must be assigned an equality type E.

There are three rules for assigning the `prop` type to an expression, which may be `T` or an equality or freshness constraint. The rule for an equality constraint $v \mathtt{=} v'$ requires that v and v' both have the same equality type E. The freshness constraint rule is similar to the abstraction rule described above and implies that the value on the left-hand side of the `#` must be a variable x of name sort.

In the rule for existentials, the newly-generated variable must be of an equality type E, so we cannot generate meta-variables to stand for unknown α-trees or unknown functions, for example. The side-condition $x \notin dom(\Gamma)$ can always be satisfied by α-renaming the variable at the meta-level. The other side-condition ($\alpha\text{-}Tree_{\Sigma}(E) \neq \emptyset$) ensures that there exists at least one potential instantiation for the meta-variable x—it would be unsound to create meta-variables ranging over an empty type. Given a datatype declaration Σ it is decidable whether any given equality type is inhabited. The family of sets $\alpha\text{-}Tree_{\Sigma}(E)$ is formalised in Definition 6 below.

The typing judgement for frame stacks has the form $\Gamma \vdash F : T \to T'$, which means that the stack accepts a value of type T and that the overall result of the computation has type T'. This judgement is defined by the final two rules from Figure 2.

2.4 Operational semantics

We now present the operational semantics of αML as a non-deterministic, small-step transition relation between abstract machine configurations. This requires the following

definition of pure αML expressions and frame stacks, which correspond to traditional functional programs.

Definition 1 (Pure expressions and frame stacks) An expression or frame stack is *pure* if it does not contain any sub-expressions of the form $\texttt{<}v\texttt{>}v'$, T, $v = v'$, $v \,\#\, v'$, $e \parallel e'$ or $\exists x\!:\!E.\, e$.

In order to present the operational semantics cleanly, we will distinguish between pure and impure configurations:

- *Pure configurations* have the form $\langle F, e\rangle$, where F and e are both pure in the sense of Definition 1.
- *Impure configurations* have the form $\exists\Delta(\overline{c}; F; e)$. The additional elements correspond to the side-effecting features of αML: the generation of new meta-variables (stored in Δ) and the collection of mutually satisfiable constraints (recorded using CLP techniques).

We define a typing relation $\Gamma \vdash \exists\Delta(\overline{c}; F; e)\!:\!T$ for impure configurations, which holds if $dom(\Gamma) \cap dom(\Delta) = \emptyset$ (always satisfiable by α-renaming) and, for some type T', we have $\Gamma, \Delta \vdash e\!:\!T'$, $\Gamma, \Delta \vdash F\!:\!T' \to T$ and $\Gamma, \Delta \vdash c\!:\!\texttt{prop}$ (for all $c \in \overline{c}$).

We now turn to the rules which define the operational semantics of αML. Figure 3 presents small-step rules for two transition relations:

- $\langle F, e\rangle \to_P \langle F', e'\rangle$ between two pure configurations; and
- $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ between two impure configurations.

The $\to_P$ relation captures the behaviour of an eager functional programming language with `case` expressions, tupling, projection and recursive functions. The properties of such languages have been extensively studied so we will not discuss the pure transition rules in great detail. The impure transition rule (I1) uses the pure transition rules to incorporate functional programming into αML, which leads to the following result concerning the evaluation of pure expressions.

Theorem 2 (Embedded functional programming language) *Suppose that the typing judgement $\emptyset \vdash \exists\emptyset(\mathsf{T}; F; e)\!:\!T$ holds. Then, if F and e are pure then $\exists\emptyset(\mathsf{T}; F; e) \longrightarrow \exists\Delta(\overline{c}; F'; e')$ holds iff $\Delta = \emptyset$, $\overline{c} = \mathsf{T}$, F' and e' are pure, and $\langle F, e\rangle \to_P \langle F', e'\rangle$.*

Proof By cases on the impure reduction rules, using the fact that purity is preserved by the $\to_P$ rules. □

Rule (I2) tests the new constraint c for satisfiability with the existing constraints in $\overline{c}$, by deciding whether $\models \exists\Delta(\overline{c}\ \&\ c)$. This satisfiability judgement is true if there exists an instantiation V of the variables in $dom(\Delta)$ (defined formally in Definition 7 below) which simultaneously satisfies c and all of the constraints in $\overline{c}$. We will not discuss constraint satisfaction here—see [19] for more details. If the constraints are mutually satisfiable then it is safe to continue—the new constraint is incorporated into the constraint environment. If $\exists\Delta(\overline{c}\ \&\ c)$ is *not* satisfiable then this branch of the computation can proceed no further.

Rule (I3) is responsible for generating new meta-variables to stand for unknown values of equality type. The side condition $x \notin dom(\Delta)$ ensures that x is indeed new, and is trivially satisfiable by meta-level α-renaming.

Pure transitions: $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$

(P1) $\langle F \circ (x.\, e), v\rangle \rightarrow_P \langle F, e[v/x]\rangle$.

(P2) $\langle F, (\texttt{let}\ x = e\ \texttt{in}\ e')\rangle \rightarrow_P \langle F \circ (x.\, e'), e\rangle$.

(P3) $\langle F, v\, v'\rangle \rightarrow_P \langle F, e[v/f, v'/x]\rangle$
if v is $\texttt{fun}\, f\,(x : T) : T' = e$.

(P4) $\langle F, (\texttt{case}\ K_i\, v\ \texttt{of}\ K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n)\rangle \rightarrow_P \langle F, e_i[v/x_i]\rangle$
if $i \in \{1, \ldots, n\}$.

(P5) $\langle F, (v_1, \ldots, v_n).\, i\rangle \rightarrow_P \langle F, v_i\rangle$
if $i \in \{1, \ldots, n\}$.

Impure transitions: $\exists \Delta(\overline{c}; F; e) \longrightarrow \exists \Delta'(\overline{c}'; F'; e')$

(I1) $\exists \Delta(\overline{c}; F; e) \longrightarrow \exists \Delta(\overline{c}; F'; e')$
if $\langle F, e\rangle \rightarrow_P \langle F', e'\rangle$.

(I2) $\exists \Delta(\overline{c}; F; c) \longrightarrow \exists \Delta(\overline{c}\ \&\ c; F; \mathsf{T})$
if $\not\models \exists \Delta(\overline{c}\ \&\ c)$.

(I3) $\exists \Delta(\overline{c}; F; \exists x : E.\, e) \longrightarrow \exists \Delta, x : E(\overline{c}; F; e)$
if $x \notin dom(\Delta)$.

(I4) $\exists \Delta(\overline{c}; F; \texttt{case}\ x\ \texttt{of}\ K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n) \longrightarrow \exists \Delta, x_i : E_i(\overline{c}\ \&\ x = K_i\, x_i; F; e_i)$
if $i \in \{1, \ldots, n\}$ and $\texttt{datatype}\ S\ =_\Sigma\ K_1\ \texttt{of}\ E_n \mid \cdots \mid K_n\ \texttt{of}\ E_n$,
where $\Delta(x) = S$ and $\not\models \exists \Delta, x_i : E_i(\overline{c}\ \&\ x = K_i\, x_i)$.

(I5) $\exists \Delta(\overline{c}; F; x.\, i) \longrightarrow \exists \Delta, x_1 : E_1, \ldots, x_n : E_n(\overline{c}\ \&\ x = (x_1, \ldots, x_n); F; x_i)$
if $i \in \{1, \ldots, n\}$ and $\Delta(x) = E_1 * \cdots * E_n$.

(I6) $\exists \Delta(\overline{c}; F; e_1 \parallel e_2) \longrightarrow \exists \Delta(\overline{c}; F; e_i)$
if $i \in \{1, 2\}$.

Fig. 3 Small-step operational semantics for αML

Rules (I4) and (I5) are the impure counterparts of rules (P4) and (P5) for deconstructing tuples and data values respectively. These rules are necessary because the ability to generate meta-variables means that a well-typed, closed expression could reduce to an expression like "$\texttt{case}\, x\, \texttt{of}\, K_1\, x_1 \texttt{->} e_1 \mid \cdots \mid K_n\, x_n \texttt{->} e_n$". Similarly, we may need to project out of an unknown tuple. Rule (I4) causes non-determinism by *narrowing* over the unknown value x. Narrowing involves non-deterministically guessing instantiations for unknown arguments to a function or `case` expression, so that the expression may be evaluated further. There is a considerable literature on this particular kind of non-determinism, largely centred round the functional logic programming language Curry—see [15] for a survey.

Rule (I6) introduces more non-deterministic branching by simply allowing the expression $e_1 \parallel e_2$ to transition either to e_1 or to e_2. No search strategy is specified, and no treatment of stuck computation branches is prescribed—such issues are not important in our theoretical treatment. There is no communication between the various branches of an αML computation.

Definition 2 We say that an αML *program* is any closed expression e. The *initial configuration* for a program e is $\exists \emptyset(\mathsf{T}; \mathtt{Id}; e)$.

We now define notions of success and failure for αML programs. Success corresponds to normal termination of a functional program, and failure corresponds to a logic program reporting that no derivation exists for the user's query. The predicate for

successful termination will feature prominently in our definition of operational equivalence in Section 4.

Definition 3 (Success) A configuration *has succeeded* if it is of the form $\exists\Delta(\overline{c}; \mathtt{Id}; v)$, where $\models \exists\Delta(\overline{c})$ holds. A configuration *may succeed*, written $\exists\Delta(\overline{c}; F; e)\downarrow$, if there exists a finite sequence of $\longrightarrow$-reductions to a configuration that has succeeded. We write $\exists\Delta(\overline{c}; F; \overline{c})\downarrow^n$ if there exists such a sequence of length less than or equal to n.

Definition 4 (Failure) A configuration *has failed* if it takes one of the following forms:

- $\exists\Delta(\overline{c}; \mathtt{Id}; v)$ where $\exists\Delta(\overline{c})$ is not satisfiable; or
- $\exists\Delta(\overline{c}; F; c')$ where $\exists\Delta(\overline{c}\ \mathtt{\&}\ c')$ is not satisfiable.

A configuration *must fail*, written $\exists\Delta(\overline{c}; F; e)$ *fails*, if every sequence of impure reductions is finite and leads to a configuration that has failed. We write $\exists\Delta(\overline{c}; F; e)$ *fails*^{n} if all such sequences are of length less than or equal to n.

We can now state the standard safety results which tell us that well-typed αML programs do not stop making impure transitions unless they have reached a state of success or failure.

Theorem 3 (Preservation of satisfaction) *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and* $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ *then* $\models \exists\Delta(\overline{c})$ *iff* $\models \exists\Delta'(\overline{c}')$. □

Theorem 4 (Type preservation) *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and there exists a configuration* $\exists\Delta'(\overline{c}'; F'; e')$ *such that* $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$*, then* $\emptyset \vdash \exists\Delta'(\overline{c}'; F'; e') : T$ *holds.* □

Theorem 5 (Progress) *If* $\emptyset \vdash \exists\Delta(\overline{c}; F; e) : T$ *and* $\exists\Delta(\overline{c}; F; e)$ *has neither succeeded nor failed, then* $\exists\Delta(\overline{c}; F; e) \longrightarrow \exists\Delta'(\overline{c}'; F'; e')$ *holds for some* $\exists\Delta'(\overline{c}'; F'; e')$. □

The proofs of these results are straightforward case analyses over the small-step transition rules from Definition 3.

3 Encoding abstract syntax trees in αML

In this section we formalise an encoding of ASTs into αML. We begin by recalling the K combinator as a useful example of a closed λ-term. As mentioned above, this is typically written as the schematic term $\lambda x.\, \lambda y.\, x$, which is intended to denote the α-equivalence class $[\mathtt{Lam}\ \mathtt{<}n\mathtt{>(Lam}\ \mathtt{<}n'\mathtt{>(Var}\ n\mathtt{))}]_\alpha$ (using the nominal signature for λ-terms introduced in Section 2.1). Since the names n and n' are assumed to follow the permutative convention, our encoding into αML (which uses NPNAS) must have the following features.

- the generation of new meta-variables to stand for names;
- constraints that these names must be distinct from each other;
- a value which represents the structure of the AST.

The rest of this section presents such an encoding, and we will recap the K combinator example at the end of the section.

We specify the valid ASTs of our object-language using a *nominal signature* Σ as described above. To model object-level bindable names we will use the permutative names $n \in \mathit{Name}$ introduced in Section 1. We assume the existence of a total function *sort* which maps every name n to a name sort N in Σ and is such that there are infinitely many names assigned to every name sort. We say that $n \in \mathit{Name}(N)$ if $\mathit{sort}(n) = N$.

Definition 5 (Ground trees) We write Tree_Σ for the set of all syntax trees over the nominal signature Σ. With names (and unit) as our building blocks, we define classes $g \in \mathit{Tree}_\Sigma(E)$ of *ground trees* of the various equality types by constructor application, tupling and name abstraction, as follows.

$$\frac{\mathit{sort}(n) = N}{n \in \mathit{Tree}_\Sigma(N)} \qquad \frac{}{\texttt{()} \in \mathit{Tree}_\Sigma(\texttt{unit})} \qquad \frac{g_1 \in \mathit{Tree}_\Sigma(E_1) \qquad \cdots \qquad g_n \in \mathit{Tree}_\Sigma(E_n)}{(g_1, \ldots, g_n) \in \mathit{Tree}_\Sigma(E_1 * \cdots * E_n)}$$

$$\frac{g \in \mathit{Tree}_\Sigma(E) \qquad (K : E \to S) \in \Sigma}{K\ g \in \mathit{Tree}_\Sigma(S)} \qquad \frac{\mathit{sort}(n) = N \qquad g \in \mathit{Tree}_\Sigma(E)}{\texttt{<}n\texttt{>}g \in \mathit{Tree}_\Sigma(\texttt{[}N\texttt{]}E)}$$

The most interesting case here is for abstractions $\texttt{<}n\texttt{>}g$, which inhabit the family of abstraction types $\texttt{[}N\texttt{]}E$. This enforces syntactically that we can only bind a single name, which must be of a name sort N. We write $\mathit{FN}(g)$ for the set of *free names* of a tree, i.e. those names n which appear without an enclosing $\texttt{<}n\texttt{>}-$ abstraction.

Definition 6 (α-trees) Our ground trees correspond precisely to the ground nominal terms of [34]—i.e. those which do not contain logic variables. Hence there is a well-studied notion of α-equivalence between ground trees, and we will write $\alpha\text{-}\mathit{Tree}_\Sigma(E)$ for the set of all α-equivalence classes $[g]_\alpha$ of ground trees $g \in \mathit{Tree}_\Sigma(E)$, which we call *$\alpha$-trees*.

Definition 7 (α-tree valuations) An *α-tree valuation* V is a finite partial function which maps variables to α-trees. We write $\mathit{dom}(V)$ for the domain of the partial function. Given a type environment Δ we write $\alpha\text{-}\mathit{Tree}_\Sigma(\Delta)$ for the set of all α-tree valuations V such that $\mathit{dom}(V) = \mathit{dom}(\Delta)$ and $V(x) \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta(x))$ for all $x \in \mathit{dom}(V)$. This ensures that the valuation respects types.

We will use α-tree valuations to instantiate αML values of equality types, which coincide with the schematic patterns of [19]. Given an αML value v such that $\Delta \vdash v : E$, we can show (using the techniques of [26]) that there exists a well-behaved notion of pattern instantiation $[\![v]\!]_V$ which produces an α-tree $[g]_\alpha \in \alpha\text{-}\mathit{Tree}_\Sigma(E)$.

We now proceed to the technical details of our AST encoding. This translation must be carefully defined so that our correctness theorem is true. By Definition 5 the only identifiers which appear in ground trees are names n, which do not appear in the syntax of αML. We therefore fix a bijection between the countably infinite sets of names (Name) and variables (Var). We write $\mathcal{V}(n)$ for the variable corresponding to the name n. This will be used to translate the free names of a ground tree. To deal with the bound names, we introduce the following technical device.

Definition 8 (Name environments) Let ε range over *name environments*, which are finite partial functions from the set of names (Name) to the set of variables (Var).

We write $\varepsilon[n \mapsto x]$ for the environment which maps n to x but otherwise behaves like ε. In particular, we will write ε_g for the environment $\{n \mapsto \mathcal{V}(n) \mid n \in FN(g)\}$ which maps the free names of g to variables according to the fixed bijection. We write $dom(\varepsilon)$ and $cod(\varepsilon)$ for the domain and codomain of ε, respectively.

The names in a ground tree are always of a name sort—hence the variables occurring in translated trees should also all be of name sort. We now pick out some important relationships between name environments and type environments η. Throughout, we write $\overline{n}$ for a finite set of distinct names. For any name environment ε we write η_ε for the corresponding typing environment $\{\varepsilon(n) : sort(n) \mid n \in dom(\varepsilon)\}$. Furthermore, we will write $\Gamma \vdash_\varepsilon \overline{n}$ to mean that $\Gamma(\varepsilon(n)) = sort(n)$ for all $n \in \overline{n}$, i.e. the type environment Γ respects the name-sorting function on the image of $\overline{n}$ under ε. For the special case where ε is the (appropriate subset of the) fixed bijection $\mathcal{V}(n)$ between names and variables, we will elide the environment and just write $\Gamma \vdash \overline{n}$.

Our translation of ground trees follows the intuition that "*what matters about names when they are used to describe binding structure is not their particular identity, but rather the distinctions between them*" [19]. Hence we must express pairwise distinctions between meta-variables (of name sort) drawn from a finite set.

Definition 9 (Name distinction constraints) Fix a set $\overline{x}$ of (distinct) variables $x_1, \ldots, x_n$. Then, define $\#_{\overline{x}}$ to be the set of atomic constraints

$$\#_{\overline{x}} \triangleq \{x_i \# x_j \mid 1 \leq i < j \leq n\}. \tag{2}$$

We now use this syntactic sugar to define the translation of ground trees into αML.

Definition 10 (Tree translation) For a ground tree g, suppose that $\varepsilon_g \vdash \langle \eta_{\varepsilon_g}, g\rangle \triangleright \langle \eta', v_g\rangle$ holds (this auxiliary relation is described below). Then, the αML translation $[\![g]\!]$ of g is given by

$$[\![g]\!] \triangleq \exists(\eta' - \eta_{\varepsilon_g}).\, \#_{dom(\eta')} \,\&\, v_g.$$

The variables in $dom(\eta')$ are all of those used in the encoding of the ground tree g (this is a property of the $\triangleright$ relation). The variables in $dom(\eta_{\varepsilon_g})$ are those which correspond to the *free names* of g. Hence, the variables in $dom(\eta' - \eta_{\varepsilon_g})$ represent the *bound names* of g. This is meaningful because the abstracted names within a particular ground tree are treated concretely.

The αML representation $[\![g]\!]$ of a ground tree g is not a value but rather an expression which, when evaluated, creates a pattern v_g that reflects the *syntactic structure* of g and generates constraints which represent the *binding structure* of g. The variables corresponding to free names of g are free variables of its encoding $[\![g]\!]$, and the variables corresponding to bound names of g are bound at the meta-level in the expression $[\![g]\!]$. The freshness constraints require that all of the names be mutually distinct, thereby modelling the permutative behaviour of the underlying set *Name*. The main theorem of Section 5 will demonstrate that, taken together, the value and the constraints faithfully represent the α-tree $[g]_\alpha$.

The auxiliary relation $\varepsilon \vdash \langle \eta, g\rangle \triangleright \langle \eta', v\rangle$ transforms a type environment η and a ground tree g into another type environment η' and an αML value v, all in the presence of a name environment ε. The type environments record the αML variables that occur in the translation of the tree, so that new variables can be generated, and the name environment ensures that the binding scope of the names from the tree is respected.

$$\frac{\varepsilon(n) = x}{\varepsilon \vdash \langle \eta, n \rangle \ \triangleright \ \langle \eta, x \rangle} \qquad \frac{}{\varepsilon \vdash \langle \eta, \mathtt{()} \rangle \ \triangleright \ \langle \eta, \mathtt{()} \rangle} \qquad \frac{\varepsilon \vdash \langle \eta, g \rangle \ \triangleright \ \langle \eta', v \rangle}{\varepsilon \vdash \langle \eta, K\ g \rangle \ \triangleright \ \langle \eta', K\ v \rangle}$$

$$\frac{\varepsilon \vdash \langle \eta, g_1 \rangle \ \triangleright \ \langle \eta_1, v_1 \rangle \qquad \cdots \qquad \varepsilon \vdash \langle \eta_{k-1}, g_k \rangle \ \triangleright \ \langle \eta', v_k \rangle}{\varepsilon \vdash \langle \eta, \mathtt{(}g_1 \mathtt{,} \ldots \mathtt{,} g_k \mathtt{)} \rangle \ \triangleright \ \langle \eta', \mathtt{(}v_1 \mathtt{,} \ldots \mathtt{,} v_k \mathtt{)} \rangle}$$

$$\frac{x \notin dom(\eta) \qquad \varepsilon[n \mapsto x] \vdash \langle \eta, x : sort(n), g \rangle \ \triangleright \ \langle \eta', v \rangle}{\varepsilon \vdash \langle \eta, \mathtt{<}n\mathtt{>}g \rangle \ \triangleright \ \langle \eta', \mathtt{<}x\mathtt{>}v \rangle}$$

Fig. 4 Tree translation rules

The $\triangleright$ relation is defined by the rules in Figure 4. The type environment is threaded through the definition as state, whereas the name environment follows the binding structure of the AST. The name environment is only used by the rule for names, which looks up a name in the environment and translates it to the appropriate variable.

The abstraction rule modifies both the type and name environments. For an abstraction $\mathtt{<}n\mathtt{>}g$, the type environment is enlarged by choosing a fresh variable which does not already occur in η and adding it, with the same type $sort(n)$. The name environment is then modified by overriding any existing mapping for n so that it is mapped to the freshly-generated variable x. This corresponds to the lexical scope of that bound occurrence of n in the original tree. There may well be multiple occurrences of n, even multiple bound occurrences: this is not problematic because each binding occurrence gets implemented by a distinct variable in αML and the environment ensures that the binding structure of the tree is faithfully represented.

Remark 2 The encoding $[g]_F$ of λ-terms into FreshML [30, Definition 3.7.2] is as follows.

$$[x]_F \triangleq \mathtt{Var}\ x \tag{3}$$

$$[\lambda x.\, t]_F \triangleq \mathtt{let}\ x\ \mathtt{=}\ \mathtt{fresh}\ \mathtt{in}\ \mathtt{Lam}\ \mathtt{<}x\mathtt{>}[t]_F \tag{4}$$

$$[t\ t']_F \triangleq \mathtt{App}\ \mathtt{(}[t]_F\ \mathtt{,}\ [t']_F\mathtt{)} \tag{5}$$

Here we re-use the nominal signature from Section 2.1 to represent λ-terms in the meta-language. The key feature to note is that λ-bound variables are represented using meta-level bound variables which are instantiated with a fresh name by the `fresh` operator. Thus the FreshML encoding of λ-terms enforces the permutative convention implicitly by its use of globally fresh names to stand for bound variables.

Our encoding of ground trees into αML defined in Definition 10 is interesting because it does not rely on the fresh generation operation which is used in the FreshML translation. The αML encoding is certainly more involved than the FreshML one because information on name distinctness must be made explicit and passed around the program. Returning to our running example of the `K` combinator, the encoding of the α-equivalence class representative $\mathtt{Lam}\ \mathtt{<}n\mathtt{>}\mathtt{(Lam}\ \mathtt{<}n'\mathtt{>}\mathtt{(Var}\ n\mathtt{))}$ in αML is

$$\exists x\mathtt{:var}.\, \exists y\mathtt{:var}.\, \mathtt{(}x\ \mathtt{\#}\ y\ \mathtt{\&}\ \mathtt{Lam}\ \mathtt{<}x\mathtt{>(Lam}\ \mathtt{<}y\mathtt{>(Var}\ x\mathtt{)))} \tag{6}$$

where we assume that $sort(n) = sort(n') = \mathtt{var}$. This translation satisfies the three criteria which we mentioned at the start of the section. Two new meta-variables, x

and y, are generated to stand for the object-level names. To model permutative behaviour we require that these two names be distinct from each other. Finally, the value $\texttt{Lam<}x\texttt{>(Lam<}y\texttt{>(Var}\,x\texttt{))}$ models the actual syntactic structure of the term.

As a further example, the αML encoding of the ground tree $\texttt{(<}n\texttt{><}n\texttt{>}n\texttt{,(}n\texttt{,}\,n'\texttt{))}$ is

$$\exists x_1 : N.\, \exists x_2 : N.\, \#_{\{x_1, x_2, \mathcal{V}(n), \mathcal{V}(n')\}}\ \texttt{\&}\ \texttt{(<}x_1\texttt{><}x_2\texttt{>}x_2\texttt{,(}\mathcal{V}(n)\texttt{,}\,\mathcal{V}(n')\texttt{))} \tag{7}$$

where we assume that $sort(n) = sort(n') = N$. This example illustrates the handling of multiple, nested occurrences of the same name in abstraction position and furthermore a clash between a bound and a free name. Repeated occurrences in binding position are handled correctly because every single binder is modelled by a newly-generated meta-variable, and free occurrences are translated using the fixed bijection $\mathcal{V}(n)$.

4 Contextual equivalence for αML

In this section we develop a theory of contextual equivalence for αML. We begin by defining a notion of *operational equivalence* between well-typed αML expressions, which is based on observing the results of successful computations of the αML abstract machine. We then show that this equivalence relation corresponds to contextual equivalence in the broader sense.

Definition 11 (Operational equivalence) Recalling the definition of success from Definition 3, we define the operational equivalence relation $\boldsymbol{\Delta \vdash e \cong e' : T}$ which holds iff $\Delta \vdash e : T$ and $\Delta \vdash e' : T$ both hold and

$$\exists \Delta'(\overline{c}; F; e)\downarrow \iff \exists \Delta'(\overline{c}; F; e')\downarrow$$

holds for all Δ', $\overline{c}$, F and T' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash \overline{c} : \texttt{prop}$ and $\Delta' \vdash F : T \to T'$.

We extend this definition to a relation $\cong^\circ$ between arbitrary αML expressions, including those which contain free variables that are not of equality types. We will refer to $\cong^\circ$ as the *open extension* of $\cong$, even though both relations contain expressions with free variables. The open extension is defined in terms of $\cong$ by substituting values (which only contain free variables of equality types) for the free variables which are not of an equality type. We write $e[\sigma]$ for the capture-avoiding application of the substitution σ to e.

Definition 12 (Open extension of $\cong$) Let the typing environment Γ be decomposed into disjoint typing environments Δ and Γ', where $\Gamma'(x)$ is not an equality type for any $x \in dom(\Gamma')$. Then, the open extension of operational equivalence $\boldsymbol{\Gamma \vdash e \cong^\circ e' : T}$ holds iff $\Delta' \vdash e[\sigma] \cong e'[\sigma] : T$ holds for all $\Delta' \supseteq \Delta$ and all capture-avoiding substitutions $\sigma \in Sub_\Sigma(\Gamma', \Delta')$, which is the set of all substitutions which map variables $x \in dom(\Gamma')$ to values $\sigma(x)$ such that $\Delta' \vdash \sigma(x) : \Gamma'(x)$.

Before we consider the properties of the operational equivalence relation defined above, we first present a general notion of type-respecting relations between αML expressions, after [25] and [27].

Definition 13 (Expression relations) An *expression relation* $\mathcal{E}$ is a set of tuples (Γ, e, e', T), made up of a typing environment, two expressions and a type, such that $\Gamma \vdash e : T$ and $\Gamma \vdash e' : T$. We write $\Gamma \vdash e\ \mathcal{E}\ e' : T$ to mean that $(\Gamma, e, e', T) \in \mathcal{E}$. We now enumerate some standard properties of expression relations. We say that

$$\frac{x \in dom(\Gamma) \qquad \Gamma(x) = T}{\Gamma \vdash x \mathrel{\widehat{\mathcal{E}}} x : T} \qquad \frac{(K : T \to D) \in \Sigma \qquad \Gamma \vdash v \mathrel{\mathcal{E}} v' : T}{\Gamma \vdash K\, v \mathrel{\widehat{\mathcal{E}}} K\, v' : D}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : T_1 \qquad \cdots \qquad \Gamma \vdash v_n \mathrel{\mathcal{E}} v_n' : T_n}{\Gamma \vdash (v_1, \ldots, v_n) \mathrel{\widehat{\mathcal{E}}} (v_1', \ldots, v_n') : T_1 * \cdots * T_n} \qquad \frac{}{\Gamma \vdash () \mathrel{\mathcal{E}} () : \mathtt{unit}}$$

$$\frac{\Gamma, f : T \to T', x : T \vdash e \mathrel{\mathcal{E}} e' : T' \qquad f, x \notin dom(\Gamma)}{\Gamma \vdash (\mathtt{fun}\, f\, (x : T) : T' = e) \mathrel{\widehat{\mathcal{E}}} (\mathtt{fun}\, f\, (x : T) : T' = e') : T \to T'} \qquad \frac{}{\Gamma \vdash \mathsf{T} \mathrel{\widehat{\mathcal{E}}} \mathsf{T} : \mathtt{prop}}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : N \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash \mathtt{<}v_1\mathtt{>}v_2 \mathrel{\widehat{\mathcal{E}}} \mathtt{<}v_1'\mathtt{>}v_2' : \mathtt{[}N\mathtt{]}\, E} \qquad \frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : E \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash (v_1 = v_2) \mathrel{\widehat{\mathcal{E}}} (v_1' = v_2') : \mathtt{prop}}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : N \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : E}{\Gamma \vdash (v_1 \mathbin{\#} v_2) \mathrel{\widehat{\mathcal{E}}} (v_1' \mathbin{\#} v_2') : \mathtt{prop}} \qquad \frac{\Gamma \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \Gamma, x : T \vdash e_2 \mathrel{\mathcal{E}} e_2' : T' \qquad x \notin dom(\Gamma)}{\Gamma \vdash (\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2) \mathrel{\widehat{\mathcal{E}}} (\mathtt{let}\, x = e_1'\, \mathtt{in}\, e_2') : T'}$$

$$\frac{\Gamma \vdash v_1 \mathrel{\mathcal{E}} v_1' : T \to T' \qquad \Gamma \vdash v_2 \mathrel{\mathcal{E}} v_2' : T}{\Gamma \vdash (v_1\, v_2) \mathrel{\widehat{\mathcal{E}}} (v_1'\, v_2') : T'}$$

$$\frac{\begin{array}{c} x_1 \neq \ldots \neq x_n \notin dom(\Gamma) \qquad D = K_1 \mathtt{\ of\ } T_1 \mid \cdots \mid K_n \mathtt{\ of\ } T_n \\ \Gamma \vdash v \mathrel{\mathcal{E}} v' : D \qquad \Gamma, x_1 : T_1 \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \cdots \qquad \Gamma, x_n : T_n \vdash e_n \mathrel{\mathcal{E}} e_n' : T \end{array}}{\begin{array}{l} \Gamma \vdash (\mathtt{case}\, v\, \mathtt{of}\, K_1\, x_1 \mathtt{\ ->\ } e_1 \mid \cdots \mid K_n\, x_n \mathtt{\ ->\ } e_n) \mathrel{\widehat{\mathcal{E}}} \\ \qquad (\mathtt{case}\, v'\, \mathtt{of}\, K_1\, x_1 \mathtt{\ ->\ } e_1' \mid \cdots \mid K_n\, x_n \mathtt{\ ->\ } e_n') : T \end{array}}$$

$$\frac{\Gamma \vdash v \mathrel{\mathcal{E}} v' : T_1 * \cdots * T_n \qquad i \in \{1, \ldots, n\}}{\Gamma \vdash (v.\, i) \mathrel{\widehat{\mathcal{E}}} (v'.\, i) : T_i} \qquad \frac{\Gamma \vdash e_1 \mathrel{\mathcal{E}} e_1' : T \qquad \Gamma \vdash e_2 \mathrel{\mathcal{E}} e_2' : T}{\Gamma \vdash (e_1 \parallel e_2) \mathrel{\widehat{\mathcal{E}}} (e_1' \parallel e_2') : T}$$

$$\frac{x \notin dom(\Gamma) \qquad \alpha\text{-}Tree_\Sigma(E) \neq \emptyset \qquad \Gamma, x : E \vdash e \mathrel{\mathcal{E}} e' : T}{\Gamma \vdash (\exists x : E.\, e) \mathrel{\widehat{\mathcal{E}}} (\exists x : E.\, e') : T}$$

Fig. 5 Compatible refinement $\widehat{\mathcal{E}}$ of an expression relation $\mathcal{E}$.

- $\mathcal{E}$ is **an equivalence relation** if it is *reflexive* ($\Gamma \vdash e : T \implies \Gamma \vdash e \mathrel{\mathcal{E}} e : T$), *symmetric* ($\Gamma \vdash e \mathrel{\mathcal{E}} e' : T \implies \Gamma \vdash e' \mathrel{\mathcal{E}} e : T$) and *transitive* ($\Gamma \vdash e \mathrel{\mathcal{E}} e' : T \wedge \Gamma \vdash e' \mathrel{\mathcal{E}} e'' : T \implies \Gamma \vdash e \mathrel{\mathcal{E}} e'' : T$).
- $\mathcal{E}$ has the **weakening** property if $\Gamma \vdash e \mathrel{\mathcal{E}} e' : T$ and $\Gamma' \supseteq \Gamma$ imply $\Gamma' \vdash e \mathrel{\mathcal{E}} e' : T$.
- $\mathcal{E}$ is **substitutive** if $\Gamma, \Gamma' \vdash e \mathrel{\mathcal{E}} e' : T$ and $\Gamma \vdash \sigma \mathrel{\mathcal{E}} \sigma' : \Gamma'$ imply $\Gamma \vdash e[\sigma] \mathrel{\mathcal{E}} e'[\sigma] : T$, where $\Gamma \vdash \sigma \mathcal{E} \sigma' : \Gamma'$ means that $\sigma, \sigma' \in Sub_\Sigma(\Gamma, \Gamma')$ and that $\Gamma' \vdash \sigma(x) \mathcal{E} \sigma'(x) : \Gamma(x)$ holds for all $x \in dom(\Gamma)$.
- $\mathcal{E}$ is **compatible** if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$, where $\widehat{\mathcal{E}}$ is the *compatible refinement* of $\mathcal{E}$ (defined in Figure 5).
- $\mathcal{E}$ is **adequate** if $\Delta \vdash e \mathrel{\mathcal{E}} e' : T$ implies $\Delta \vdash e \cong e' : T$.

Most of these definitions are fairly standard. The most interesting is compatibility, which states that membership of the expression relation is preserved by the term-formers of the αML language. A property of expression relations that is stated explicitly in [27] but omitted from Definition 13 is equivariance—it is trivial to show that all αML expression relations are equivariant because names do not appear in the syntax of αML.

We note that operational equivalence ($\cong^\circ$) is an expression relation because it requires that the two expressions both have the same type. The following theorem enu-

merates some desirable properties of $\cong^\circ$. We call it CIU after [21], because in Definition 12 we quantify over all possible closing substitutions before testing the termination behaviour of expressions.

Theorem 6 (CIU) *Operational equivalence, $\cong^\circ$, is an equivalence relation and has the weakening property. Furthermore, it is adequate, substitutive and compatible. It is also the largest such expression relation.*

Proof It is easy to show that $\cong^\circ$ is an equivalence relation and has the weakening and adequacy properties. We now give brief outlines of the proofs of the other properties.

- **$\cong^\circ$ is substitutive**
  It suffices to prove the case where the typing environment Γ maps all variables in $dom(\Gamma)$ to equality types, and where the substitutions are both singletons, i.e. we aim to show that

$$\Delta, x{:}T \vdash e \cong^\circ e'{:}T' \ \wedge\ \Delta \vdash v \cong v'{:}T \implies \Delta \vdash e[v/x] \cong e'[v'/x]{:}T'.$$

  This is sufficient because we can repeatedly apply this result to simulate any closing substitution, including those used to define $\cong^\circ$ in terms of $\cong$. We therefore assume that $\Delta, x{:}T \vdash e \cong^\circ e'{:}T'$ and $\Delta \vdash v \cong v'{:}T$ both hold. By choosing appropriate configurations we may infer that

$$\exists \Delta'(\overline{c}; F; e[v'/x])\downarrow \iff \exists \Delta'(\overline{c}; F; e'[v'/x])\downarrow \tag{8}$$
$$\exists \Delta'(\overline{c}; F \circ (x.\, e); v)\downarrow \iff \exists \Delta'(\overline{c}; F \circ (x.\, e); v')\downarrow \tag{9}$$

  both hold, where $\Delta' \supseteq \Delta$, $\Delta' \vdash \overline{c}{:}\mathtt{prop}$ and $\Delta' \vdash F{:}T' \to T''$ all hold, for some Δ', $\overline{c}$, F and T''. Then, using rules (I1) and (P1) we can make a $\longrightarrow$ transition on both sides of (9) to get that $\exists \Delta'(\overline{c}; F; e[v/x])\downarrow \iff \exists \Delta'(\overline{c}; F; e[v'/x])\downarrow$ holds. Then, we can combine this with (8) to get $\exists \Delta'(\overline{c}; F; e[v/x])\downarrow \iff \exists \Delta'(\overline{c}; F; e'[v'/x])\downarrow$, and finally by Definition 12 we get that $\Delta \vdash e[v/x] \cong e'[v'/x]{:}T'$ holds, as required.
- **$\cong^\circ$ is compatible**
  Since $\cong^\circ$ is known to be reflexive and substitutive, it suffices to prove this result for the special case where the expressions e and e' only contain variables of equality types. Therefore we must show that $\Delta \vdash e \mathrel{\widehat{\cong^\circ}} e'{:}T$ implies $\Delta \vdash e \cong^\circ e'{:}T$.
  We use an operational proof technique similar to that of [27], involving a variant of Howe's method [16]. Although this technique is powerful, the reasons why it works are somewhat mysterious. For this reason, we proved our compatibility result by case analysis on the compatible extension $\widehat{\cong^\circ}$ of the operational equivalence relation, rather than by a single large termination induction as in [27, Appendix A]. Although more long-winded, this approach to proving compatibility draws attention to the particular case where we require the full power of Howe's method—the case for recursive function values. The other cases are dealt with by choosing appropriate αML evaluation contexts, as in the substitutivity proof. The reader is referred to [18, Appendix A] for the details of the compatibility proof.
- **$\cong^\circ$ is the largest expression relation with the above properties**
  We fix an expression relation $\mathcal{E}$, and assume that $\mathcal{E}$ is reflexive, symmetric and transitive, adequate, substitutive and compatible, and that $\mathcal{E}$ has the weakening property. Now, we must show that $\mathcal{E} \subseteq \cong^\circ$. Suppose that $\Gamma \vdash e\ \mathcal{E}\ e'{:}T$ holds, and that $\Gamma = \Delta, \Gamma'$, where $\Gamma'(x)$ is not an equality type for all $x \in dom(\Gamma')$. Now, we pick an arbitrary substitution $\sigma \in Sub_\Sigma(\Gamma', \Delta')$, for some $\Delta' \supseteq \Delta$. Since $\mathcal{E}$ is

reflexive we know that $\Gamma' \vdash \sigma \, \mathcal{E} \, \sigma : \Delta'$ holds. Then, by the fact that $\mathcal{E}$ is substitutive and has the weakening property, we get that $\Delta' \vdash e[\sigma] \, \mathcal{E} \, e'[\sigma] : T$ holds, and since $\mathcal{E}$ is adequate we know that $\Delta' \vdash e[\sigma] \cong e'[\sigma] : T$. Finally, by definition of $\cong^\circ$, this is equivalent to $\Gamma \vdash e \cong^\circ e' : T$, as required.

This completes the proof of Theorem 6. □

Following the approach of [14] and [20], we have therefore shown that the $\cong^\circ$ relation coincides with the standard notion of contextual equivalence, $\Gamma \vdash e \cong_{\mathrm{ctx}} e' : T$ which holds iff $\Gamma \vdash e : T$ and $\Gamma \vdash e' : T$ both hold and

$$\forall \mathcal{C} \in \mathit{Ctx}_{\Sigma}(T). \; \mathcal{C}[e]{\downarrow} \iff \mathcal{C}[e']{\downarrow}.$$

Here, $\mathit{Ctx}_{\Sigma}(T)$ is the set of all αML program contexts $\mathcal{C}$ which accept values of type T (the definition is straightforward given the language syntax defined in Figure 1). The CIU theorem shows that $\cong^\circ$ possesses the key properties of contextual equivalence, being the largest congruence relation which is adequate. Henceforth we will refer to $\cong^\circ$ as *contextual equivalence*.

5 Correctness of the encoding

We now show that two ground trees g and g' are α-equivalent precisely when their encodings $[\![g]\!]$ and $[\![g']\!]$ are contextually equivalent in αML.

Theorem 7 (Correctness of representation for αML) *If $\eta \vdash \mathit{FN}(g, g')$ then $g =_\alpha g' : E$ holds if and only if $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$.*

This is a fundamental correctness result if we claim to support meta-programming with binders handled correctly modulo α-equivalence. A similar result was proved for FreshML [30,27] but the proof presented here is substantially different because the language constructs of αML are different to those of FreshML. We will prove the two directions separately in the remainder of this section. Given the definition of $[\![g]\!]$ in Definition 10, we can immediately dispense with the forward direction.

Lemma 1 ($=_\alpha$ implies $\cong$) *If $g =_\alpha g' : E$ and $\eta \vdash \mathit{FN}(g, g')$ then $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$.*

Proof If $g =_\alpha g' : E$ then g and g' differ only by an α-renaming of their abstracted variables. However, in the translations $[\![g]\!]$ and $[\![g']\!]$ these variables are bound by $\exists$-quantifiers. Since we identify αML expressions up to α-conversion, it follows that $[\![g]\!]$ and $[\![g']\!]$ are in fact the same expression. Then, since the tree translation preserves types we have $\eta \vdash [\![g]\!] : E$, and since $\cong$ is reflexive it follows that $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$, as required. □

The other direction is less straightforward, for two reasons: because names do not appear in the syntax of αML and because the representation of a ground tree in αML is an expression, not a value. However, the proof is not as complicated as that for FreshML, since the αML language contains built-in constructs for the solving of equality constraints, which actually involves checking whether values are α-equivalent. Therefore, if we know that two translated trees are contextually equivalent, we know something about their operational behaviour when placed in contexts that are capable of testing α-equivalence. The following proof exploits this capability of the αML meta-language.

Definition 14 Given a set of names $\overline{n}$, a name environment ε with $dom(\varepsilon) \supseteq \overline{n}$ and a valuation $V \in \alpha\text{-}Tree_{\Sigma}(\eta_{\varepsilon})$ such that $V \models \#_{\varepsilon(\overline{n})}$, we write $\pi_{V,\varepsilon,\overline{n}}$ to stand for some (equivalently, any) permutation such that $V(\varepsilon(n)) = \{\pi_{V,\varepsilon,\overline{n}}(n)\}$ for all $n \in \overline{n}$.

Since names are not present in the syntax of αML, we will relate an αML expression $[\![g]\!]$ with the original ground tree g *up to a permutation*. This is related to the idea of unifying up to a permutation from equivariant unification [5]. Using Definition 14, we will prove intermediate results about the relationship between a tree g and the constraints $\overline{c}_g$ and value v_g produced by evaluating its αML representation $[\![g]\!]$, leading up to the key lemma that contextual equivalence implies α-equivalence for ground trees.

Lemma 2 *If* $g \in [\![v]\!]_V$ *and* $\eta \vdash v : E$ *and* $V(x) = \{n\}$ *then* $\texttt{<}n\texttt{>}g \in [\![\texttt{<}x\texttt{>}v]\!]_V$.

Proof This follows from a general result about the existence of a pattern instantiation operation $[\![v]\!]_V$ which respects α-equivalence, using techniques developed in [26]. □

We now use Lemma 2 to prove that the result of evaluating a tree expression is equivalent (up to a permutation) to the tree itself. This is a central lemma in relating the operational behaviour of translated trees to their α-equivalence. We observe that the explicit freshness information is required so that the permutation $\pi_{V,\varepsilon,\overline{n}}$ is guaranteed to exist (see Definition 14).

Lemma 3 *If* $dom(\varepsilon) = \overline{n}$, $\overline{n} \supseteq FN(g)$, $\eta \vdash_{\varepsilon} \overline{n}$ *and* $\varepsilon \vdash \langle \eta, g \rangle \ \triangleright \ \langle \eta', v \rangle$ *then, for all* $V \in \alpha\text{-}Tree_{\Sigma}(\eta')$, *if* $V \models \#_{dom(\eta')}$ *then for some (or any) permutation* $\pi_{V,\varepsilon,\overline{n}}$ *it is the case that* $(\pi_{V,\varepsilon,\overline{n}} \cdot g) \in ([\![v_g]\!]_V)$.

Proof The proof is by induction on the structure of g. The cases which do not involve names are straightforward: the tuple case merely relies on the fact that the $\triangleright$ relation never discards any information from the type environment. The case when g is a name n follows from the definition of $\varepsilon(n)$ and from that of the permutation $\pi_{V,\varepsilon,\overline{n}}$. The case for an abstraction $\texttt{<}n\texttt{>}g'$ proceeds by induction using the abstraction rule from Figure 4. This case relies on Lemma 2 and other standard facts about permutations and freshness from nominal logic, such as [34, Lemma 2.8]. We omit the detail of the calculations in the interest of brevity. □

We now relate the operational behaviour of a translated ground tree $[\![g]\!]$ back to the ground tree g itself. The following lemma shows that evaluating $[\![g]\!]$ produces constraints $\overline{c}_g$ and a value v_g which faithfully represent the structure of g, in the sense that any valuation which satisfies $\overline{c}_g$ can be applied to v_g to produce an α-equivalence class which can be obtained from g by the application of a permutation (as mentioned above).

Lemma 4 *If* $\emptyset \vdash \exists \Delta(\overline{c}; F; [\![g]\!]) : T$ *and* $\models \exists \Delta(\overline{c} \ \& \ \#_{dom(\eta_{\varepsilon_g})})$ *both hold then*

$$\exists \Delta(\overline{c}; F; [\![g]\!]) \longrightarrow \cdots \longrightarrow \exists \Delta, \eta_g(\overline{c} \ \& \ \overline{c}_g; F; v_g)$$

and, for all $V \in \alpha\text{-}Tree_{\Sigma}(\Delta, \eta_g)$, *if* $V \models (\overline{c} \ \& \ \overline{c}_g)$ *then for some (or any) permutation* $\pi_{V,\varepsilon_g,(FN(g))}$ *it is the case that* $(\pi_{V,\varepsilon_g,(FN(g))} \cdot g) \in [\![v_g]\!]_V$.

Proof This follows from Lemma 3 and the transition rules of αML. □

We can now use Lemma 4 to prove the main result of this section, where we write $\#_{\varepsilon}$ as a shorthand for the mutual freshness constraints $\#_{cod(\varepsilon)}$.

Lemma 5 ($\cong$ implies $=_\alpha$) *If $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$ and $\eta \vdash FN(g, g')$ then $g =_\alpha g' : E$.*

Proof We assume that $\eta \vdash [\![g]\!] \cong [\![g']\!] : E$ holds. This means that $[\![g]\!]$ and $[\![g']\!]$ have identical termination behaviour in any given (well-typed) context. Writing ε^* for the result of merging the environments ε_g and $\varepsilon_{g'}$, we observe that $cod(\varepsilon^*) \subseteq dom(\eta)$. We now focus on the behaviour of the following two configurations.

1. $\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\ y = [\![g']\!]\ \mathtt{in}\ z = y); [\![g']\!])$
2. $\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\ y = [\![g']\!]\ \mathtt{in}\ z = y); [\![g]\!])$

From contextual equivalence we know that the termination behaviour of configurations 1 and 2 are identical. Thus it suffices to show firstly that configuration 1 terminates, and secondly that if configuration 2 terminates then $g =_\alpha g' : E$ holds. The proofs of these follow.

– **Configuration 1 terminates.**
Since $\eta \vdash FN(g, g')$ we know that $\models \exists\eta(\#_{\varepsilon^*})$ holds. Using Lemma 4 we can show that

$$\begin{aligned}&\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\ y = [\![g']\!]\ \mathtt{in}\ z = y); [\![g']\!])\\ &\quad\longrightarrow \cdots \longrightarrow \exists\eta, \eta_1, \eta_2(\#_{\varepsilon^*}\ \&\ \overline{c}_1\ \&\ \overline{c}_2; \mathtt{Id}; v_1 = v_2)\end{aligned}$$

holds (after some constraint simplification). We can also show that there exists a valuation $V \in \alpha\text{-}Tree_\Sigma(\eta, \eta_1, \eta_2)$ such that

$$(\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') \in [\![v_1]\!]_V \tag{10}$$
$$(\pi_{V,\varepsilon^*,(FN(g'))} \cdot g') \in [\![v_2]\!]_V \tag{11}$$

both hold. Now, in order to show that configuration 1 terminates, it suffices to show that $\models \exists\eta, \eta_1, \eta_2(\#_{\varepsilon^*}\ \&\ \overline{c}_1\ \&\ \overline{c}_2\ \&\ v_1 = v_2)$. We have already shown that $V \models \#_{\varepsilon^*}\ \&\ \overline{c}_1\ \&\ \overline{c}_2$, so it remains only to see that $V \models v_1 = v_2$. This follows from (10) and (11).

– **If configuration 2 terminates then $g =_\alpha g' : E$.**
We assume that configuration 2 terminates. Thus we know that

$$\begin{aligned}&\exists\eta(\#_{\varepsilon^*}; \mathtt{Id} \circ (z.\,\mathtt{let}\ y = [\![g']\!]\ \mathtt{in}\ z = y); [\![g]\!])\\ &\longrightarrow \cdots \longrightarrow \exists\eta, \eta_g, \eta_{g'}(\#_{\varepsilon^*}\ \&\ \overline{c}_g\ \&\ \overline{c}_{g'}\ \&\ v_g = v_{g'}; \mathtt{Id}; \mathsf{T})\end{aligned}$$

holds (also after some constraint simplification) where η_g, $\overline{c}_g$ and v_g are the results of evaluating $[\![g]\!]$ and similarly $\eta_{g'}$, $\overline{c}_{g'}$ and $v_{g'}$ were produced by evaluation of $[\![g']\!]$. Furthermore, we know that the final constraint is satisfiable, i.e. that

$$\models \exists\eta, \eta_g, \eta_{g'}(\#_{\varepsilon^*}\ \&\ \overline{c}_g\ \&\ \overline{c}_{g'}\ \&\ v_g = v_{g'}).$$

From this we get that there exists $V \in \alpha\text{-}Tree_\Sigma(\eta, \eta_g, \eta_{g'})$ such that $V \models \#_{\varepsilon^*}$ and $[\![v_g]\!]_V = [\![v_{g'}]\!]_V$ both hold (as well as $V \models \overline{c}_g$ and $V \models \overline{c}_{g'}$). Using Lemma 3 we can show that $(\pi_{V,\varepsilon^*,\overline{n}^*} \cdot g) =_\alpha (\pi_{V,\varepsilon^*,\overline{n}^*} \cdot g') : E$. Finally, by [34, equation 9] we can eliminate the permutations from both sides to leave $g =_\alpha g' : E$, as required.

This completes the proof of Lemma 5. □

The correctness theorem for αML (Theorem 7) follows from Lemma 1 and Lemma 5. This result is novel because our representation of ground trees is correct up to α-equivalence but does not rely on the existence of globally-fresh names and a mechanism to dynamically generate these. Such features are central to existing encodings, such as the FreshML example presented in Remark 2 above. The αML encoding scheme revolves around *explicit*, *local* freshness of names (because names are not assumed to be distinct unless stated otherwise) whereas the FreshML scheme exploits *implicit*, *global* freshness (because names are always assumed to be distinct). It is of theoretical interest that both approaches allow encodings encoding of ASTs involving binding which are provably correct in the sense of Theorem 1 (for FreshML) and Theorem 7 (for αML).

Remark 3 (Contextual equivalence and finite failure) We can define a finer notion of operational equivalence for αML where the terms must have identical success behaviour *and* identical failure behaviour (call this $\cong_F^\circ$). This relation is of interest because it is clearly possible to observe the difference between divergence and finite failure in practice. Furthermore, $\cong_F^\circ$ has many of the same good properties as $\cong^\circ$. In particular, a version of the correctness result (Theorem 7) also holds when $\cong_F^\circ$ is used as the notion of program equivalence. This is true essentially because the evaluation of an encoded ground tree $[\![g]\!]$ never diverges, so observing successful termination is equivalent to observing both succesful termination and finite failure. There are situations when evaluation of $[\![g]\!]$ could fail finitely, for example we have

$$\exists\{\mathcal{V}(n_1)\texttt{:var}, \mathcal{V}(n_2)\texttt{:var}\}(\mathcal{V}(n_1)\texttt{ = }\mathcal{V}(n_2); \texttt{Id}; \texttt{App (Var}\, n_1\texttt{ , Var}\, n_2\texttt{)})\ \mathit{fails}$$

because the implicit constraint $\mathcal{V}(n_1)\texttt{ \# }\mathcal{V}(n_2)$ is unsatisfiable in conjunction with the explicit constraint that $\mathcal{V}(n_1)\texttt{ = }\mathcal{V}(n_2)$. However, Theorem 7 still holds because if $\exists\Delta(\overline{c}; F; [\![g]\!])$ *fails* and $g =_\alpha g' : E$ then $\exists\Delta(\overline{c}; F; [\![g']\!])$ *fails* also, since g and g' have the same sets of free names.

6 Related and future work

6.1 Nominal meta-programming

We have already discussed FreshML [30] at length, and αProlog [4] in passing, as examples of closely related nominal meta-programming systems. The work on the theory of contextual equivalence for FreshML [31,27] is the most closely related work to that described in this paper. In particular, the operationally-based proof strategy of [27] greatly influenced our work.

That paper also investigated the effect of adding operations on bindable names, such as a linear order, on the behavioural properties of contextual equivalence. Similar extensions to αML would most likely take the form of an extended grammar of constraints. For example, we might add a constraint $x < x'$ between variables x and x' of name sort. This would be satisfied by any valuation V such that $V(x) < V(x')$ holds, for some linear order $<$ on *Name*. We cite this specific example because a linear order on bindable names has practical applications, such as the efficient implementation of type environments as balanced binary trees.

Thanks to the modular design of αML, adding new flavours of constraint is fairly straightforward. However, the main correctness results proved in this paper (Theorem 6 and Theorem 7) are sensitive to such details and would need to be re-checked.

In particular, the proof that operational equivalence is *compatible* depends on the semantics of constraints. Ideally, the proof would be independent of such details. We could achieve this by characterising those properties of the constraint language which are required in order for the various correctness results to be true. This would yield a version of αML parameterised by a constraint domain, along much the same lines as existing CLP languages [17].

The proofs described in this paper are promising candidates for mechanisation, perhaps using the Nominal Isabelle [32] theorem proving system. In particular, the proofs of the CIU result (Theorem 6) are long and tedious, making them ideal for machine-assisted verification.

6.2 Fresh name generation

In [19, Remark 4.3] we noted that it is possible to extend the core αML language with a fresh name generation primitive similar to that from FreshML, without moving away from the "names as meta-variables" design philosophy. The syntax is extended with an operation $\texttt{fresh}\ N$ of type N, which has the following impure reduction rule.

$$\text{(I7)}\quad \exists \Delta(\overline{c}; F; \texttt{fresh}\ N) \longrightarrow \exists \Delta, x{:}N(\overline{c}\ \texttt{\&}\ x\ \texttt{\#}\ \Delta; F; x) \qquad \text{where } x \notin dom(\Delta).$$

In the definition of rule (I7) we write $x\ \texttt{\#}\ \Delta$ for the constraint $x\ \texttt{\#}\ x_1\ \texttt{\&}\ \cdots\ \texttt{\&}\ x\ \texttt{\#}\ x_n$, where $dom(\Delta) = \{x_1, \ldots, x_n\}$. The fresh name is returned as a newly-generated meta-variable, together with a set of new freshness constraints which require that it be fresh for every other meta-variable generated so far. These constraints make explicit the implicit convention of FreshML that distinct names are fresh for each other and that the newly-generated name may not be free in any unknown object-level data terms. This is sufficient to define a generative unbinding operator [27] which mimics that of FreshML.

It is straightforward to rework the theory of contextual equivalence for αML developed in Section 4 for the extended language with fresh name generation. This gives us the mathematical tools necessary to study whether `fresh` is definable up to contextual equivalence in the core language, as discussed in [19, Remark 4.3]. We believe that `fresh` cannot be defined compositionally in terms of core αML, because rule (I7) requires run-time introspection of Δ. However, we have been unable to find a proof either way. Hence the relative expressiveness of the language with and without `fresh` remains an open question.

It may be possible to translate programs which use `fresh` into core αML via a whole-program transformation. Such a translation could use a monadic programming style to pass the list of generated variables around the program as explicit state so that the fresh name generation operation can be implemented using the operations available in αML. An expression e of type T involving `fresh` would be translated to a core αML expression $\ulcorner e \urcorner$ of type $\texttt{env} \to \texttt{(env,}\ T\texttt{)}$, where `env` is some type which records the names generated so far (e.g. a list of names). The translation would thread the current environment of generated names through the execution so that translated occurrences of `fresh` in the program can access it explicitly.

We do not have a proof that a correct transformation exists: this is left for future work. Ideally we would relate the result of evaluating a program involving `fresh` to the result of evaluating its core αML translation and show that the translation preserves contextual equivalence.

6.3 Complex binding structures

Our presentation has only dealt with the binding of a *single* name using the $\texttt{<}v\texttt{>}v'$ term-former. We chose to focus on this because of its simplicity and expressiveness. Other languages offer more flexible primitives for expressing binding structures, such as Cαml [28] and `ott` [29]. We believe that it may be possible to model more complex binding structure in αML using the combination of single name binding and explicit freshness constraints. For example, consider a mutually-recursive `letrec` construct in the object-language.

$$\texttt{letrec}\ f_1\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ f_2\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e$$

For simplicity we will assume that there are just two mutually recursive functions. If we extended our nominal signature with a `LetRec` constructor with the appropriate types, this could be encoded as an αML expression something like

$$\begin{array}{l}\texttt{let}\ z_1\ \texttt{=}\ [\![e_1]\!]\ \texttt{in}\ \texttt{let}\ z_2\ \texttt{=}\ [\![e_2]\!]\ \texttt{in}\ \texttt{\#}_{\{f_1,f_2,x_1,x_2\}}\ \texttt{\&}\\ x_1\ \texttt{\#}\ z_2\ \texttt{\&}\ x_2\ \texttt{\#}\ z_1\ \texttt{\&}\ \texttt{LetRec}\ \texttt{(<}f_1\texttt{><}f_2\texttt{>(}[\![e]\!]\texttt{,<}x_1\texttt{><}x_2\texttt{>(}z_1\texttt{,}\ z_2\texttt{)))}\end{array}$$

where we write $[\![e^*]\!]$ for the encoding of e^*, where f_1, x_1, f_2 and x_2 are pairwise distinct and where z_1 and z_2 are distinct, suitably fresh variables. The structure of the expession ensures that f_1 and f_2 are both bound in e and in the bodies of the two functions. To express that x_1 is only bound in e_1 and that x_2 is only bound in e_2, we bind both x_1 and x_2 in e_1 *and* e_2 then add freshness constraints so that x_1 cannot appear free in e_2 and x_2 cannot appear free in e_1. This is similar to the use of the `inner` and `outer` keywords for specifying binding scope in Cαml [28].

To our knowledge there are no concrete mathematical results concerning the relative expressiveness of different methods for representing binding structure in a meta-language. We believe that the approach outlined above could allow Cαml-style binding structures to be encoded into αML, but we do not have a proof. This is certainly worthy of further investigation.

6.4 Adequacy theorems in logical frameworks

Outside the world of nominal methods, the most closely related results are the adequacy theorems for logical frameworks. These are succinctly explained by the following quotation from [23].

> *Adequeacy theorems play a critical role in logical frameworks They guarantee that we can translate expressions from the object-language in the meta-language, compute with them, and then interpret the results back in the object-language. ... They ensure that formal reasoning in the logical framework is correct with respect to the object logic under construction.*

Logical frameworks tend to use higher-order abstract syntax (HOAS) techniques to represent object-level binding. The meta-language is then a simply-typed λ-calculus and binding in the object-language is encoded using meta-level λ-abstraction. A standard notion of single capture-avoiding substitution then comes for free using meta-level β-reduction. This is an elegant solution from the user's perspective because the problem of dealing with binders is left as a problem to be solved at the meta-level.

As mentioned above, adequacy results for logical frameworks are geared towards ensuring that *reasoning* at the meta-level is sound with respect to the object-level. Concerns with object-language α-equivalence are dismissed by appealing to α-conversion in the meta-level, which is a significant departure from our approach. Also, since the notion of equivalence at the meta-level is usually $\alpha\beta\eta$-equivalence, the issues with contextual equivalence that we confronted in this paper do not arise either. It makes more sense to talk about contextual equivalence in the setting of a programming language than a logical framework.

7 Conclusion

We have developed a theory of contextual equivalence for the αML meta-language which was introduced in [19]. We have used our notion of contextual equivalence to prove the key correctness result that ASTs with binding structure can be faithfully encoded in the meta-language so that contextual equivalence and α-equivalence coincide. This result is not obvious because αML omits certain features such as fresh name generation which were central to the corresponding proofs for FreshML [31, 27]. Therefore we have shown that using non-permutative meta-variables to represent binders, along with local constraints expressed distinctness between names, suffices to encode terms with binding correctly.

References

1. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics, revised edn. North-Holland (1984)
2. de Bruijn, N.: Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae **34**, 381–392 (1972)
3. Cheney, J.: The complexity of equivariant unification. In: J. Díaz, J. Karhumäki, A. Lepistö, D. Sannella (eds.) Proceedings of the 31$^{\text{st}}$ International Colloquium on Automata, Languages and Programming (ICALP 2004), *Lecture Notes in Computer Science*, vol. 3142, pp. 332–344. Springer-Verlag (2004)
4. Cheney, J.: Nominal logic programming. Ph.D. thesis, Cornell University (2004)
5. Cheney, J.: Equivariant unification. In: J. Giesl (ed.) Proceedings of the 16$^{\text{th}}$ International Conference on Rewriting Techniques and Applications (RTA 2005), *Lecture Notes in Computer Science*, vol. 3467, pp. 74–89. Springer-Verlag (2005)
6. Cheney, J., Urban, C.: Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In: B. Demoen, V. Lifschitz (eds.) Proceedings of the 20$^{\text{th}}$ International Conference on Logic Programming (ICLP 2004), no. 3132 in Lecture Notes in Computer Science, pp. 269–283. Springer-Verlag (2004)
7. Cheney, J., Urban, C.: Nominal logic programming. ACM Transactions on Programming Languages and Systems **30**(5), 1–47 (2008)
8. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science **103**(2), 235–271 (1992)
9. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1993), *ACM SIGPLAN Notices*, vol. 28, pp. 237–247. ACM Press (1993)
10. Gabbay, M.J.: A theory of inductive definitions with α-equivalence: semantics, implementation, programming language. Ph.D. thesis, University of Cambridge (2000)
11. Gabbay, M.J., Mathijssen, A.: Capture-avoiding substitution as a nominal algebra. Formal Aspects of Computing **20**(4–5), 451–479 (2008)

12. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing **13**(3–5), 341–363 (2002)
13. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science **13**(2), 225–230 (1981)
14. Gordon, A.D.: Operational equivalences for untyped and polymorphic object calculi. In: A.D. Gordon, A.M. Pitts (eds.) Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, pp. 9–54. Cambridge University Press (1998)
15. Hanus, M.: Multi-paradigm declarative languages. In: V. Dahl, I. Niemelä (eds.) Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), *Lecture Notes in Computer Science*, vol. 4670, pp. 45–75. Springer-Verlag (2007)
16. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Information and Computation **124**(2), 103–112 (1996)
17. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: Semantics of constraint logic programming. Journal of Logic Programming **37**(1–3), 1–46 (1998)
18. Lakin, M.R.: An executable meta-language for inductive definitions with binders. Ph.D. thesis, University of Cambridge (2010)
19. Lakin, M.R., Pitts, A.M.: Resolving inductive definitions with binders in higher-order typed functional programming. In: G. Castagna (ed.) Proceedings of the 18th European Symposium on Programming (ESOP 2009), *Lecture Notes in Computer Science*, vol. 5502, pp. 47–61. Springer-Verlag (2009)
20. Lassen, S.B.: Relational reasoning about contexts. In: A.D. Gordon, A.M. Pitts (eds.) Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, pp. 91–135. Cambridge University Press (1998)
21. Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. Journal of Functional Programming **1**(3), 287–327 (1991)
22. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press (1997)
23. Pfenning, F.: Logical frameworks. In: A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, chap. 17, pp. 1063–1147. Elsevier Science and MIT Press (2001)
24. Pitts, A.M.: Operational semantics and program equivalence. In: Applied Semantics, Advanced Lectures, *Lecture Notes in Computer Science*, vol. 2395, pp. 378–412. Springer-Verlag (2002)
25. Pitts, A.M.: Typed operational reasoning. In: B.C. Pierce (ed.) Advanced Topics in Types and Programming Languages, chap. 7, pp. 245–289. MIT Press (2005)
26. Pitts, A.M.: Alpha-structural recursion and induction. Journal of the ACM **53**(3), 459–506 (2006)
27. Pitts, A.M., Shinwell, M.R.: Generative unbinding of names. Logical Methods in Computer Science **4**(1:4), 1–33 (2008)
28. Pottier, F.: An overview of Cαml. In: N. Benton, X. Leroy (eds.) Proceedings of the 2005 ACM-SIGPLAN Workshop on ML (ML 2005), *Electronic Notes in Theoretical Computer Science*, vol. 148, pp. 27–52. Elsevier (2006)
29. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. In: R. Hinze, N. Ramsey (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), pp. 1–12. ACM Press (2007)
30. Shinwell, M.R.: The Fresh Approach: functional programming with names and binders. Ph.D. thesis, University of Cambridge (2005)
31. Shinwell, M.R., Pitts, A.M.: On a monadic semantics for freshness. Theoretical Computer Science **342**(1), 28–55 (2005)
32. Urban, C.: Nominal Techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4), 327–356 (2008)
33. Urban, C., Cheney, J.: Avoiding equivariance in Alpha-Prolog. In: P. Urzyczyn (ed.) Proceedings of the 7th International Conference on Typed Lambda Calculus and Applications (TLCA 2005), no. 3461 in Lecture Notes in Computer Science, pp. 74–89. Springer-Verlag (2005)
34. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoretical Computer Science **323**(1–3), 473–497 (2004)