JFP **23** (6): 658–700, 2013.
doi:10.1017/S0956796813000245 First published online 13 December 2013

Contextual equivalence for inductive definitions with binders in higher order typed functional programming

MATTHEW R. LAKIN* and ANDREW M. PITTS

Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
(*e-mail:* {Matthew.Lakin,Andrew.Pitts}@cl.cam.ac.uk)

Abstract

Correct handling of names and binders is an important issue in meta-programming. This paper presents an embedding of constraint logic programming into the αML functional programming language, which provides a provably correct means of implementing proof search computations over inductive definitions involving names and binders modulo α-equivalence. We show that the execution of proof search in the αML operational semantics is sound and complete with regard to the model-theoretic semantics of formulae, and develop a theory of contextual equivalence for the subclass of αML expressions which correspond to inductive definitions and formulae. In particular, we prove that αML expressions, which denote inductive definitions, are contextually equivalent precisely when those inductive definitions have the same model-theoretic semantics. This paper is a revised and extended version of the conference paper (Lakin, M. R. & Pitts, A. M. (2009) Resolving inductive definitions with binders in higher-order typed functional programming. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*, Castagna, G. (ed), Lecture Notes in Computer Science, vol. 5502. Berlin, Germany: Springer-Verlag, pp. 47–61) and draws on material from the first author's PhD thesis (Lakin, M. R. (2010) *An Executable Meta-Language for Inductive Definitions with Binders*. University of Cambridge, UK).

1 Introduction

Many important meta-programming tasks, such as the implementation of compilers and theorem provers, may be specified in terms of relations defined inductively using a finite set of schematic inference rules. Furthermore, it is common that the object-languages involved will contain some flavour of names and name-binding in their syntax. For example, in the λ-calculus the variable x is assumed to be bound in the term t in the term $\lambda x.t$ and, by convention, terms are considered equal up to capture-avoiding renaming of these bound variables (Barendregt, 1984).

Assuming that such a specification in terms of inductively defined relations has been given, the next task is to produce an implementation that meets the specification. If the specification is encoded in a formal meta-language, it may be executed directly or compiled into executable code, thereby providing an enhanced degree of confidence in the

* Current address: Department of Computer Science, University of New Mexico, Albuquerque, NM 87131-0001, USA.

correctness of the resulting computations. Ideally, the meta-language and its associated compiler should also handle the thorny issues of names and binding automatically. As a practical matter, it is desirable to encode inductively defined relations in a syntax which is as close as possible to the language of informal mathematics, thereby enabling rapid prototyping of inductive definitions involving binders.

In this paper, we use the αML meta-programming language as our host language for the specification and evaluation of inductive definitions over syntax trees involving binders. The αML meta-language was introduced in Lakin & Pitts (2009) and described in detail in Lakin (2010). It is a higher order functional-logic programming language which incorporates certain features of constraint logic programming (CLP). This allows proof-search computations over inductive definitions to be defined in a convenient way, following the informal practices of structural operational semantics. Crucially, αML uses nominal techniques to allow topologies of name-binding in object-language terms to be succinctly described in definitions, and only manipulates them such that the underlying notion of object-language α-equivalence is respected.

The contributions of this paper are: (1) to define a general encoding of inductive definitions over object-language terms involving bindable names, (2) to encode these in the αML metalanguage, (3) to prove soundness and completeness of formula evaluation, and (4) to prove contextual equivalence of semantically equivalent formulae and inductive definitions. These contributions allow inductively defined relations to be specified in a formal language that corresponds closely to executable code, allowing the behaviour of certain interesting cases of the definition to be investigated quickly and easily. While αML includes the standard features of an eager functional programming language, we note that the class of inductively defined relations considered here are constructed using a well-defined subset of the αML syntax that manipulates object-language syntax trees directly.

The remainder of this paper is organized as follows. Section 2 presents background nominal abstract syntax, the αML programming language and a corresponding notion of contextual equivalence. In Section 3 we introduce our formal model of inductive definitions over object-language terms with binders. We outline an encoding of inductive definitions into αML in Section 4, along with soundness and completeness results. In Section 5 we prove that schematic formulae and inductive definitions are contextually equivalent precisely when they have the same semantics. We outline related work in Section 6 and conclude with a discussion in Section 7.

2 Background

We begin with some background on data representation using non-permutative nominal abstract syntax (Lakin, 2011), and on the αML programming language (Lakin & Pitts, 2009) and its associated notion of contextual equivalence (Lakin & Pitts, 2012).

2.1 Background on nominal signatures and equality types

We consider abstract syntax trees specified using nominal signatures (Urban *et al.*, 2004), which extend first-order algebraic signatures with a term-former that denotes the binding of a single name in the given scope.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Definition 2.1 (*Nominal signatures and equality types*)

A nominal signature Σ consists of: (1) a finite set $\mathbb{N}_\Sigma$ of *name sorts*, ranged over by N; (2) a finite set $\mathbb{S}_\Sigma$ of *nominal data sorts*, disjoint from $\mathbb{N}_\Sigma$ and ranged over by S and (3) a finite set $\mathbb{C}_\Sigma$ of *constructors* $K : E \to S$, where the argument type E is an *equality type* of Σ, generated by the following grammar,

$$E \in Ety_\Sigma ::= S \mid N \mid [N]E \mid E * \cdots * E \mid \mathtt{unit}$$

The non-standard elements here are the the name sorts N, which are inhabited by object-language names, and name abstraction sorts $[N]E$, which are inhabited by object-language terms where a single name of sort N is bound in a term of type E. We refer to these as *abstractions* because they are not treated as binders at the meta-level but are simply used to model object-language binders, as in FreshML (Shinwell *et al.*, 2003). We use the phrase *equality type* in the sense of Standard ML (Milner *et al.*, 1997), to mean types whose values admit a decidable notion of equality. Henceforth, we assume the existence of a nominal signature Σ. We now begin a running example which will illustrate our approach throughout the paper.

Example 2.2 (*A nominal signature for the untyped* λ*-calculus*)

The grammar for untyped λ-terms t (Barendregt, 1984) consists of variables (x), applications ($t\,t$) and λ-abstractions ($\lambda x.e$). In our example nominal signature Λ for untyped λ-terms we have a single name sort `var` for variables, a single nominal data sort `term` for λ-terms and three constructors corresponding to the three clauses in the grammar. Hence, $\mathbb{N}_\Lambda \equiv \{\mathtt{var}\}$, $\mathbb{S}_\Lambda \equiv \{\mathtt{term}\}$ and

$$\mathbb{C}_\Lambda \equiv \{\mathtt{Var} : \mathtt{var} \to \mathtt{term},\ \mathtt{App} : \mathtt{term} * \mathtt{term} \to \mathtt{term},\ \mathtt{Lam} : [\mathtt{var}]\mathtt{term} \to \mathtt{term}\}.$$

The crucial constructor is `Lam`, whose argument type is the abstraction type $[\mathtt{var}]\mathtt{term}$, which we will populate with abstraction terms representing the object-language binding construct.

2.2 Background on ground trees and α-equivalence classes

We let n range permutatively (Gabbay & Mathijssen, 2008) over a countably infinite set *Name* of bindable object-language names. We assume the existence of a total function *sort* which maps every name n to a name sort $N \in \mathbb{N}_\Sigma$ such that infinitely many names are assigned to every name sort. We say that $n \in Name(N)$ if $sort(n) = N$.

Definition 2.3 (*Ground trees*)

We write $Tree_\Sigma$ for the set of all syntax trees over the nominal signature Σ, which is defined by the following grammar:

$$g \in Tree_\Sigma ::= n \mid () \mid (g, \ldots, g) \mid K\,g \mid {<}n{>}g$$

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

We also define classes $Tree_\Sigma(E)$ of syntax trees of various equality types, as follows:

$$\frac{sort(n) = N}{n \in Tree_\Sigma(N)} \qquad \frac{}{() \in Tree_\Sigma(\mathtt{unit})} \qquad \frac{g_1 \in Tree_\Sigma(E_1) \qquad \cdots \qquad g_n \in Tree_\Sigma(E_n)}{(g_1,\ldots,g_n) \in Tree_\Sigma(E_1 * \cdots * E_n)}$$

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

These ground trees correspond precisely to the ground nominal terms of Urban *et al.* (2004). The abstraction term-former denotes the object-level binding of a single name, but is not a binder at the meta-level. Hence, if $n \neq n'$ then $<n>n$ and $<n'>n'$ are distinct ground trees. Therefore, we require a separate notion of α-equivalence on ground trees, as these α-equivalence classes represent the object-language terms quotiented by α-equivalence so frequently used in informal mathematical parlance.

Definition 2.4 (*α-equivalence and α-trees*)
We write $g =_\alpha g' : E$ for the congruence relation induced on pairs of ground trees (of the same equality type) by the standard notion of α-renaming of $(<n>-)$-bound names in ground trees, defined formally in Figure 1 of Pitts (2006). Let $\alpha\text{-}Tree_\Sigma(E)$ be the set of all $=_\alpha$-equivalence classes of ground trees of type E, which we call *α-trees*, and let t range over these. If $g \in Tree_\Sigma(E)$, then write $[g]_\alpha$ for the set $\{g' \mid g =_\alpha g' : E\}$ of all ground trees which are $=_\alpha$-equivalent to g.

If $g \in Tree_\Sigma(E)$ then $[g]_\alpha \in \alpha\text{-}Tree_\Sigma(E)$. From the rules in Definition 2.3, it also follows that if $t \in \alpha\text{-}Tree_\Sigma(N)$ then $t = [n]_\alpha = \{n\}$ for some $n \in Name(N)$. We now present standard notions of the free names of a ground term and freshness of a name for a ground term.

Definition 2.5 (*Free names and freshness*)
Suppose that $g \in Tree_\Sigma(E)$. Then we write $FN(g)$ for the finite set of names that occur *free* in g, which we define recursively as follows:

$$FN(n) \equiv \{n\} \qquad FN(()) \equiv \{\} \qquad FN((t_1,\ldots,t_n)) \equiv \bigcup_{i\in\{1,\ldots,n\}} FN(t_i)$$

$$FN(K\,t) \equiv FN(t) \qquad FN(<n>t) \equiv FN(t) - \{n\}$$

This operation respects α-equivalence, so for an α-tree $t \in \alpha\text{-}Tree_\Sigma(E)$ we let $FN(t)$ stand for the free names $FN(g)$ of some/any ground tree $g \in Tree_\Sigma(E)$ such that $t = [g]_\alpha$. Furthermore, if $t \in \alpha\text{-}Tree_\Sigma(N)$ and $t' \in \alpha\text{-}Tree_\Sigma(E)$ then we know that $t = [n]_\alpha$ for some $n \in Name(N)$, and we write $t \# t'$ and say "t is fresh for t'" if and only if $n \notin FN(t')$.

Example 2.6 (*An α-tree corresponding to a closed λ-term*)
Recalling the nominal signature Λ for untyped λ-terms from Example 2.2, we consider the representation of particular λ-terms as α-trees. Consider the closed λ-term $\lambda x.\,\lambda y.\,x$, also known as the K combinator. The corresponding α-tree, $t_{\mathsf{K}} \in \alpha\text{-}Tree_\Lambda(\mathtt{term})$, is as follows, where we assume that n and n' are distinct names:

$$t_{\mathsf{K}} = [\mathtt{Lam}{<}n{>}(\mathtt{Lam}{<}n'{>}(\mathtt{Var}\,n))]_\alpha$$

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$
\begin{array}{rcll}
T \in Ty_\Sigma & ::= & E & \text{(equality type)} \\
& | & D & \text{(data sort)} \\
& | & \texttt{prop} & \text{(type of semi-decidable propositions)} \\
& | & T \to T & \text{(function type)} \\
& | & (T,\ldots,T) & \text{(tuple type)} \\
& | & \texttt{unit} & \text{(unit type)} \\
v \in Val_\Sigma & ::= & x,f & \text{(variable)} \\
& | & K\,v & \text{(constructor application)} \\
& | & () & \text{(unit)} \\
& | & (v,\ldots,v) & \text{(tuple)} \\
& | & \texttt{fun}\, f(x:T):T = e & \text{(recursive function)} \\
& | & \mathsf{T} & \text{(success)} \\
& | & <x>v & \text{(name abstraction)} \\
c \in Constr_\Sigma & ::= & v = v & \text{(equality constraint)} \\
& | & x\#v & \text{(freshness constraint)} \\
e \in Exp_\Sigma & ::= & v & \text{(value)} \\
& | & \texttt{let}\, x = e\, \texttt{in}\, e & (\texttt{let}\text{ binding}) \\
& | & v\,v & \text{(function application)} \\
& | & \texttt{case}\, v\, \texttt{of}\, K\,x \texttt{->} e \mid \cdots \mid K\,x \texttt{->} e & (\texttt{case}\text{ expression}) \\
& | & v.i & \text{(projection)} \\
& | & c & \text{(constraint)} \\
& | & \exists x:E.\,e & \text{(existential)} \\
& | & e \parallel e & \text{(non-deterministic branch)} \\
F \in Stk_\Sigma & ::= & \texttt{Id} & \text{(empty frame stack)} \\
& | & F \circ (x.e) & \text{(non-empty frame stack).}
\end{array}
$$

Fig. 1. αML types, values, constraints, expressions and frame stacks.

This α-tree is the α-equivalence class containing all ground trees with similar structure and binding topology to the K combinator. Hence, α-trees are an appropriate universe of discourse for our definitions of inductive definitions involving binders, and for meta-programming over these.

2.3 Background on the αML programming language

The αML programming language was described in detail in Lakin (2010). We note that the language does not include explicit support for representing α-trees or inductive definitions, and we will demonstrate encodings of these into αML below. We fix a countably infinite set *Var* of variables, ranged over by x. For a nominal signature Σ we write Ty_Σ, Val_Σ, $Constr_\Sigma$, Exp_Σ and Stk_Σ for the sets of αML types, values, constraints, expressions and frame stacks respectively. These are defined in Figure 1. We restrict the class of valid αML expressions to a form similar to the A-normal form of Flanagan *et al.* (1993), where evaluation order is specified using `let` bindings, and identify expressions up to α-conversion of bound variables. Much of the αML syntax is standard for functional programming languages: the novel constructs are the abstraction value for representing object-level binders, equality and freshness constraints, generation (and binding) of fresh existential variables of equality types and the non-deterministic branching operator.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

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

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash e' : T}{\Gamma \vdash e \parallel e' : T} \qquad \frac{x \notin dom(\Gamma) \qquad \Sigma \vdash E \; inhab \qquad \Gamma, x : E \vdash e : T}{\Gamma \vdash \exists x : E.e : T}$$

Fig. 2. Typing rules for the non-standard elements of αML syntax.

Figure 2 presents selected typing αML rules. We let Γ range over finite partial functions from *Var* to Ty_Σ, and write $dom(\Gamma)$ for the domain of Γ. We will use Δ for the special case of a type environment which maps all variables in its domain to equality types, that is, a finite partial function from *Var* to Ety_Σ. The αML type grammar is stratified so that the equality types defined in Definition 2.1 are included as a delimited subset of all αML types, and αML datatype declarations extend nominal signatures by adding more general data sorts whose constructors may take values of arbitrary types, not just equality types. We write $\Sigma \vdash E \; inhab$ to mean that the equality type E is inhabited (i.e. non-empty) under the datatype declaration Σ: this can be decided by checking for cyclic dependencies between nominal data sorts with no base case (see Section 3.3.2 of Lakin, 2010). Disallowing empty equality types is important as it would be unsound to generate existential variables ranging over an empty type. We note that typechecking and typeability are equivalent (due to the presence of explicit-type annotations in the syntax) and decidable, and that inferred types are unique.

Constraint satisfaction is fundamental to the operational semantics of αML. An *α-tree constraint problem* has the form $\exists\Delta(\overline{c})$, where $\overline{c} \equiv c_1 \,\&\, \cdots \,\&\, c_n$ for some finite collection of atomic constraints as in Figure 1. We write $\Gamma \vdash \exists\Delta(\overline{c})\; ok$ to mean that $dom(\Gamma) \cap dom(\Delta) = \emptyset$ and that $\Gamma, \Delta \vdash c : \mathtt{prop}$ holds for all $c \in \overline{c}$. Then we say that $\exists\Delta(\overline{c})$ is *satisfiable*, written $\models \exists\Delta(\overline{c})$, if there exists a mapping from the variables in $dom(\Delta)$ to α-trees so that all constraints in $\overline{c}$ are simultaneously satisfied – see Section 3 for formal definitions of constraint satisfaction.

We write NonPermSat for the decision problem $\{(\Delta, \overline{c}) \mid \models \exists\Delta(\overline{c})\}$, which we can solve using the algorithm from Lakin (2011). In a previous work (Lakin, 2010, 2011) we showed that NonPermSat is in fact NP-complete, as is equivariant unification (Cheney, 2010). Intuitively, NonPermSat is NP-complete because the ability to alias variables in abstraction position produces exponentially many topologies of aliasing as the depth of the abstractions increases. However, exponential branching in the constraint solver is only an issue as the number of nested abstractions increases *without any constraints between the bound names*. In many systems of interest, for example the λ-calculus, every bound name is assumed to be fresh, and it was shown in Section 6.4 of Lakin (2010) that this constraint problems in this subclass can be decided in polynomial time. From a practical perspective, the advantage of using this constraint problem to drive the operational semantics of αML is that we retain a conceptually simple constraint problem which is sufficiently powerful to handle many possible topologies of aliasing between variables in abstraction position.

Definition 2.7 (*Value substitutions*)
A *value substitution* $\sigma \in Sub_\Sigma(\Gamma, \Gamma')$ is a finite partial function that maps each variable $x \in dom(\sigma) = dom(\Gamma)$ to a value $\sigma(x) \in Val_\Sigma$ such that $\Gamma' \vdash \sigma(x) : \Gamma(x)$. We write $(-)[\sigma]$

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

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 $\models \exists\Delta(\overline{c}\,\&\,c)$.
(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 $\models \exists\Delta, x_i : E_i(\overline{c}\,\&\,x = K_i\, x_i)$.
(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.

for the *simultaneous*, *capture-avoiding* substitution of $\sigma(x)$ for all free occurrences of x in $(-)$, for all $x \in dom(\sigma)$. It is straightforward to show that αML typing judgements are preserved by substitution.

Figure 3 presents the operational semantics of αML. We distinguish between *pure* configurations $\langle F, e\rangle$ which only contain information relevant to the evaluation of standard pure functional programs, and *impure* configurations $\exists\Delta(\overline{c}; F; e)$. The intuitive reading of $\exists\Delta(\overline{c}; F; e)$ is that the expression e contains the (bound) existentially quantified variables from Δ, and is to be evaluated subject to the constraints from $\overline{c}$ with a continuation encoded as the frame stack F (Pitts, 2002), which were defined in Figure 1. For a closed αML expression e, the *initial configuration* has the form $\exists\emptyset(\emptyset; \texttt{Id}; e)$. We define a typing judgement $\Gamma \vdash \exists\Delta(\overline{c}; F; e) : T$ for impure configurations by the following inference rule:

$$\frac{dom(\Gamma) \cap dom(\Delta) = \emptyset \qquad \Gamma, \Delta \vdash e : T' \qquad \Gamma, \Delta \vdash F : T' \rightarrow T \qquad \forall c \in \overline{c}.\ \Gamma, \Delta \vdash c : \texttt{prop}}{\Gamma \vdash \exists\Delta(\overline{c}; F; e) : T}$$

In the interest of space we focus our discussion on the novel impure reduction rules. Rule (I2) processes a constraint when it is encountered during the execution of the abstract machine, testing it for mutual satisfiability with the constraints in $\overline{c}$ (the current αML implementation uses the algorithm from Lakin (2011) for this). Rule (I3) generates fresh

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

existential variables, and rules (I4) and (I5) are the impure counterparts of rules (P4) and (P5), which use constraint-solving to deconstruct unknown data values and tuples by *narrowing*, which involves non-determistically "guessing" partial instantiations for such unknown values. There is considerable literature on narrowing – see Hanus (2007) for a survey. In rule (I4), each branch represents a single possible narrowing instantiation with a different head constructor, and it follows that precisely one of these branches will succeed. In rule (I5), branching and constraint-solving are not required as there is only one tuple constructor. Finally, rule (I6) explicitly introduces non-determinism at branching expressions.

Definition 2.8 (*Success and failure of αML configurations*)
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.

A configuration *has failed* if it has the form $\exists\Delta(\overline{c};\mathtt{Id};v)$, where $\exists\Delta(\overline{c})$ is not satisfiable, or $\exists\Delta(\overline{c};F;c')$ where $\exists\Delta(\overline{c}\ \&\ c')$ is not satisfiable. A configuration *must fail*, which we write as $\exists\Delta(\overline{c};F;e)$ *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.

It is straightforward to show that for a given well-typed αML, configuration will either succeed, fail or diverge. See Lakin (2010) for more straightforward safety properties of the operational semantics: preservation of constraint satisfaction, as well as standard type preservation and progress results. Finally, we note that this semantics underspecifies certain aspects of the run-time behaviour of αML expressions, in particular with regard to the implementation of branching in proof-search computations. In practice the search strategy would need to be fair for the results proved below to be valid, but this is beyond the scope of the current paper.

2.4 Background on contextual equivalence in αML

We begin by defining an *operational equivalence* relation between two well-typed expressions of the same type, in which successful termination in (well-typed) configurations $\exists\Delta(\overline{c};F;-)$ is observed. Note that this definition applies to expressions whose only free variables are existentially quantified variables of equality types.

Definition 2.9 (*Operational equivalence*)
The operational equivalence relation $\Delta \vdash e \cong e' : T$ 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 $\Delta' \supseteq \Delta$, $\Delta' \vdash F : T \to T'$ and $\forall c \in \overline{c}.\ \Delta' \vdash c : \mathtt{prop}$.

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. This *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.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$
\begin{aligned}
x_C &::= [\cdot] \mid x \\
v_C &::= [\cdot] \mid x_C \mid K\,v_C \mid (v_C,\ldots,v_C) \mid \texttt{fun}\,f(x:T):T = e_C \mid <x_C>v_C \\
c_C &::= [\cdot] \mid v_C = v_C \mid x_C \,\#\, v_C \\
e_C &::= [\cdot] \mid v_C \mid \texttt{let}\,x = e_C\ \texttt{in}\ e_C \mid v_C\,v_C \mid \texttt{case}\,v_C\,\texttt{of}\,K\,x \texttt{->} e_C \mid \cdots \mid K\,x \texttt{->} e_C \mid \\
&\quad\ v_C.i \mid c_C \mid \exists x:E.e_C \mid e_C \parallel e_C
\end{aligned}
$$

Fig. 4. αML program contexts. The grammar is stratified to match that of Figure 1.

Definition 2.10 (*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, $\Gamma \vdash e \cong^\circ e' : T$, holds iff $\Delta' \vdash e[\sigma] \cong e'[\sigma] : T$ holds for all $\Delta' \supseteq \Delta$ and all $\sigma \in Sub_\Sigma(\Gamma', \Delta')$.

Many basic properties of the operational equivalence relation defined above have been explored in Lakin (2010), so we will not discuss them at length. However, it is important to relate this notion of operational equivalence to the standard notion of *contextual equivalence*, which requires equivalent termination behaviour when the expressions in question are grafted into a "hole" in a program context. Program contexts for αML can be defined as in Figure 4, which mirrors the αML grammar from Figure 1, augmenting each syntactic category to include the option of a hole (except in meta-level binding positions). We write $e_C[e]$ for the expression which results from replacing all occurrences of the hole in e_C with e, noting that care must be taken to ensure that the resulting expression is syntactically well formed (e.g. that an expression has not been inserted where a value is required). We will only consider well-formed results of context instantation. We write $\Gamma \vdash_{[\cdot]:T'} e_C : T$ for the typing relation which mirrors the standard αML typing rules for the program context syntax from Figure 4, with an additional rule that assigns the type T' to $[\cdot]$ whenever it is located, and it is not hard to show that if $\Gamma \vdash_{[\cdot]:T'} e_C : T$ and $\Gamma \vdash e : T'$ then $\Gamma \vdash e_C[e] : T$. Then we can define a contextual equivalence relation $\cong^\circ_{\text{ctx}}$ between well-typed αML expressions in terms of operational equivalence as follows:

$$
\begin{aligned}
\Gamma \vdash e \cong^\circ_{\text{ctx}} e' : T \quad \Longleftrightarrow \quad & \Gamma \vdash e : T \wedge \Gamma \vdash e' : T \wedge \\
& (\forall e_C, T'.\ \Gamma \vdash_{[\cdot]:T} e_C : T' \Longrightarrow \Gamma \vdash e_C[e] \cong^\circ e_C[e'] : T')
\end{aligned}
$$

However, for reasons that have been discussed elsewhere (Pitts, 2005, 2011), we choose not to deal directly with contextual equivalence as stated above. In particular, the possibly capturing grafting operation is uncomfortably concrete and may result in different expressions depending on the choice of bound variables, since different free variables in the grafted expression may be captured. Instead, we adopt a more abstract, relational approach that focuses on the key properties of a contextual equivalence relation. Following the techniques of Gordon (1998) and Lassen (1998), we characterize contextual equivalence as the largest type-respecting relation on αML expressions which is a congruence and contains the operational equivalence relation defined in Definition 2.10 above. The following theorem demonstrates that the $\cong^\circ$ relation satisfies these criteria, following the approach of Mason & Talcott (1991).

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

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

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

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

$$\frac{\Gamma \vdash x \cong^\circ x' : N \qquad \Gamma \vdash v \cong^\circ v' : E}{\Gamma \vdash <x>v \mathrel{\widehat{\cong^\circ}} <x'>v' : [N]E} \qquad \frac{\Gamma \vdash v_1 \cong^\circ v_1' : E \qquad \Gamma \vdash v_2 \cong^\circ v_2' : E}{\Gamma \vdash (v_1 = v_2) \mathrel{\widehat{\cong^\circ}} (v_1' = v_2') : \mathtt{prop}}$$

$$\frac{\Gamma \vdash x \cong^\circ x' : N \qquad \Gamma \vdash v \cong^\circ v' : E}{\Gamma \vdash (x \# v) \mathrel{\widehat{\cong^\circ}} (x' \# v') : \mathtt{prop}} \qquad \frac{\Gamma \vdash v_1 \cong^\circ v_1' : T \to T' \qquad \Gamma \vdash v_2 \cong^\circ v_2' : T}{\Gamma \vdash (v_1\, v_2) \mathrel{\widehat{\cong^\circ}} (v_1'\, v_2') : T'}$$

$$\frac{\Gamma \vdash e_1 \cong^\circ e_1' : T \qquad \Gamma, x : T \vdash e_2 \cong^\circ e_2' : T' \qquad x \notin dom(\Gamma)}{\Gamma \vdash (\mathtt{let}\, x = e_1\ \mathtt{in}\, e_2) \mathrel{\widehat{\cong^\circ}} (\mathtt{let}\, x = e_1'\ \mathtt{in}\, e_2') : T'}$$

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

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

$$\frac{x \notin dom(\Gamma) \qquad \Sigma \vdash E\ inhab \qquad \Gamma, x : E \vdash e \cong^\circ e' : T}{\Gamma \vdash (\exists x : E.\, e) \mathrel{\widehat{\cong^\circ}} (\exists x : E.\, e') : T}$$

Fig. 5. The compatible refinement $\widehat{\cong^\circ}$ of the αML operational equivalence relation $\cong^\circ$.

Theorem 2.11 (*CIU*)
The $\cong^\circ$ relation is the largest equivalence relation between αML expressions of the same type which has the following properties:

- **adequacy:** $\Delta \vdash e \cong^\circ e' : T$ implies $\Delta \vdash e \cong e' : T$,
- **substitutivity:** $\Gamma, \Gamma' \vdash e \cong^\circ e' : T$ and $\Gamma \vdash \sigma \cong^\circ \sigma' : \Gamma'$ imply $\Gamma \vdash e[\sigma] \cong^\circ e'[\sigma'] : T$, where $\Gamma \vdash \sigma \cong^\circ \sigma' : \Gamma'$ means that $\sigma, \sigma' \in Sub_\Sigma(\Gamma, \Gamma')$ and $\forall x \in dom(\Gamma).\ \Gamma' \vdash \sigma(x) \cong^\circ \sigma'(x) : \Gamma(x)$ both hold.
- **compatibility:** $\Gamma \vdash e \mathrel{\widehat{\cong^\circ}} e' : T$ implies $\Gamma \vdash e \cong^\circ e' : T$, where $\widehat{\cong^\circ}$ is the *compatible refinement* of $\cong^\circ$, defined in Figure 5.

Proof
It is obvious from the definition that $\cong^\circ$ is an equivalence relation, and that it is adequate. The main technical challenges are to show substitutivity and compatibility. In particular, proving compatibility is somewhat involved, and is typically achieved using a transitive closure-like construction due to Howe (1996), as used by Pitts & Shinwell (2008) and Pitts

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

(2011). Proving that $\cong^\circ$ is the largest such relation is also fairly straightforward, and we refer the interested reader to Lakin (2010) for a full proof of all of these results. □

The CIU theorem implies that the $\cong^\circ$ relation coincides with the standard notion of contextual equivalence which we denoted as $\cong^\circ_{\text{ctx}}$ above. To see why, the key is the relationship between compatible refinement and the notion of grafting terms into the holes in a program context: the structures of the compatible refinement rules from Figure 5 mirror the grammar of program contexts from Figure 4. The fact that $\widetilde{\cong^\circ}$ is contained within $\cong^\circ$ means that operational equivalence is preserved by all of the term-formers of the αML language, and it follows that if $\Gamma \vdash e \cong^\circ e' : T$ then $\Gamma \vdash e_C[e] \cong^\circ e_C[e'] : T'$ holds for any e_C such that $\Gamma \vdash_{[\cdot]:T} e_C : T'$, and hence $\Gamma \vdash e \cong^\circ_{\text{ctx}} e' : T$ holds. The reverse direction follows immediately by considering the program context which is just an empty hole. Thus, we conclude that $\Gamma \vdash e \cong^\circ e' : T \iff \Gamma \vdash e \cong^\circ_{\text{ctx}} e' : T$, and henceforth we will simply refer to $\cong^\circ$ as *contextual equivalence*.

2.5 Background on encoding ground trees in αML

An important property of the αML contextual equivalence relation, which we will exploit below, is the existence of an encoding of ground trees g into αML expressions $[\![g]\!]$ that is correct in the following sense.

Property 2.12 (*Correct encoding of ground trees into αML*)
A translation $[\![-]\!]$ of ground trees into αML satisfies the *fundamental correctness property* of αML if $g \in \alpha\text{-}Tree_\Sigma(E)$ and $g' \in \alpha\text{-}Tree_\Sigma(E)$ imply that $g =_\alpha g' : E \iff \Delta \vdash [\![g]\!] \cong [\![g']\!] : E$, for all type environments Δ which respect the sorts of the translated names. Formally, we require that $\Delta \vdash FN(g_1, \ldots, g_n)$ holds, which means that $\Delta(\mathscr{V}(n)) = sort(n)$ for all names $n \in \bigcup_{i\in\{1,\ldots,n\}} FN(g_i)$, where $\mathscr{V}$ is a fixed bijection between the sets *Name* of object-level names and *Var* of meta-level variables.

The fixed bijection $\mathscr{V}$ is used to systematically translate the free names of the ground tree into free variables of the corresponding αML expression, and abstracted names in the ground tree are bound in the αML expression. For the purpose of this paper, we will assume the existence of a translation function $[\![-]\!]$ which satisfies Property 2.12. The reader is referred to Lakin & Pitts (2012) for the full definition of the translation function and for details of the correctness theorems.

3 α-inductive definitions

We now formalise *α-inductive definitions*, which are inductively defined relations between α-equivalence classes of terms with binders. These terms correspond to the abstract syntax trees modulo α-equivalence, and the relations correspond to judgements defined over those syntax trees. These relations will be defined by schematic rules, which constitute a template for creating specific rule instances by instantiating their variables with α-equivalence classes of ground terms.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$\frac{x \in dom(\Delta) \qquad \Delta(x) = E}{\Delta \vdash x : E} \qquad \frac{\Delta \vdash p : E \qquad (K : E \to S) \in \Sigma}{\Delta \vdash K\,p : S} \qquad \frac{}{\Delta \vdash () : \mathtt{unit}}$$

$$\frac{\Delta \vdash p_1 : E_1 \qquad \cdots \qquad \Delta \vdash p_n : E_n}{\Delta \vdash (p_1, \ldots, p_n) : E_1 * \cdots * E_n} \qquad \frac{\Delta \vdash x : N \qquad \Delta \vdash p : E}{\Delta \vdash <x>p : [N]E}$$

Fig. 6. Typing rules for schematic patterns.

3.1 Syntax of α-inductive definitions

We now define the syntax of schematic formulae, inference rules and complete α-inductive definitions. Our schematic rules will be constructed from patterns, which serve as templates whose variables may be instantiated with α-equivalence classes of ground terms according to particular instantiation rules, to produce a (potentially infinite) set of ground instances. We use the fixed, countably infinite set *Var* of αML variables as placeholders for unknown α-equivalence classes of terms (i.e. unknown α-trees), ranged over by various meta-variables, typically x, y etc. These are the building blocks of our language of schematic patterns.

Definition 3.1 (*Schematic patterns*)
The set Pat_Σ of *schematic patterns* over a nominal signature Σ is defined by the following grammar:

$$p \in Pat_\Sigma ::= x \mid () \mid (p, \ldots, p) \mid K\,p \mid <x>p$$

As with ground trees, we do not treat the abstraction term-former as a binder in patterns. In order to assign types to schematic patterns we must first provide types for all the variables contained therein. Let $vars(p)$ stand for the set of all variables occurring in a pattern p. We let Δ range over finite partial functions from *Var* to Ety_Σ, which assign equality types to finitely many variables, and let $dom(\Delta)$ stand for the set of all variables in the domain of definition of Δ. Figure 6 presents rules which define a typing judgement $\Delta \vdash p : E$. Note that patterns of a name sort are a special case – if $\Delta \vdash p : N$ then it follows that $p = x$ for some variable such that $x \in dom(\Delta)$ and $\Delta(x) = N$.

We now describe the instantiation of schematic patterns, which produces specific α-trees by instantiating the variables in the pattern with α-trees.

Definition 3.2 (α*-tree valuations*)
An α*-tree valuation* V is a finite partial function which maps variables to α-trees. We write $dom(V)$ for the domain of the partial function V. Given a type environment Δ, we write $\alpha\text{-}Tree_\Sigma(\Delta)$ for the set of all α-tree valuations V such that $dom(V) = dom(\Delta)$ and $V(x) \in \alpha\text{-}Tree_\Sigma(\Delta(x))$ for all $x \in dom(V)$. This stipulates that the valuation respects types.

Using the proof techniques developed in Pitts (2006) we can show that there exists a pattern instantiation operation $[\![p]\!]_V$ which respects both types and α-equivalence classes. If $\Delta \vdash p : E$, then for every $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ there exists a function $[\![-]\!]_V$ which maps p to

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$[\![p]\!]_V \in \alpha\text{-}Tree_\Sigma(E)$ such that the following all hold:

$$\begin{array}{rcl} [\![x]\!]_V & = & V(x) \\ [\![p]\!]_V = [g]_\alpha \implies [\![K\,p]\!]_V & = & [K\,g]_\alpha \\ [\![()]\!]_V & = & [()]_\alpha \\ [\![p_1]\!]_V = [g_1]_\alpha \wedge \cdots \wedge [\![p_n]\!]_V = [g_n]_\alpha \implies [\![(p_1,\ldots,p_n)]\!]_V & = & [(g_1,\ldots,g_n)]_\alpha \\ V(x) = [n]_\alpha \wedge [\![p]\!]_V = [g]_\alpha \implies [\![<x>p]\!]_V & = & [<n>g]_\alpha \end{array}$$

It is worth noting that since the variables in patterns stand for unknown α-trees, not unknown ground trees, an instantiated pattern $p \in \alpha\text{-}Tree_\Sigma(E)$ produces an α-tree. This reflects the common practice of leaving α-equivalence implicit and using representatives to stand for the whole α-equivalence class, as stated in Convention 2.1.13 from Barendregt (1984). Since only variables x appear in patterns, and not permutative names n, our approach furthermore reflects the informal practice of using schematic variables to range over names, as in Barendregt. Since the $(<n>-)$ abstraction term-former is not a binder in the meta-language, the above operation of applying a valuation to a pattern in the object-language is a possibly capturing form of substituion with regard to meta-level names and the $(<n>-)$ abstraction term-former. This also reflects common practice when instantiating the meta-variables in schematic rules, and is a key reason why we do not identify patterns up to α-renaming of abstracted variables. For example, given distinct variables x, y, z we cannot regard the patterns $<x>z$ and $<y>z$ as equivalent because if $n \neq n'$ then the valuation $V = \{x \mapsto [n]_\alpha, y \mapsto [n']_\alpha, z \mapsto [n]_\alpha\}$ gives

$$[\![<x>z]\!]_V = [<n>n]_\alpha \neq [<n'>n]_\alpha = [\![<y>z]\!]_V .$$

This example highlights a key distinction between names and schematic variables – if we have two distinct names n_1, n_2 then these will always be distinct, whereas two distinct schematic variables x_1, x_2 could be instantiated with the same name n by a valuation, which we call *aliasing*. This approach is more general, but we can also model names which behave permutatively by imposing additional constraints that the variables must be mutually distinct. Thus, it makes no sense to define the "free" names (or variables) of a pattern.

3.2 Schematic formulae and rules

We now use the language of schematic patterns from the previous section to develop a language of schematic formulae and inductive rules. Suppose that we wish to define n mutually recursive relations. We will fix *relation symbols* $r_1,\ldots,r_n$ with associated equality types $E_1,\ldots,E_n$, and write $r_i \subseteq E_i$ to mean that E_i is the equality type associated with r_i. Generally speaking, a particular schematic rule from an inductive definition of such relations might take the form

$$\frac{r_j\,p_j \;\cdots\; r_k\,p_k \qquad c_1 \cdots\; c_m}{r_i\,p_i}$$

where $i,j,k \in \{1,\ldots,n\}$. The conclusion of this rule is an atomic formula where $r_i \subseteq E_i$, and where $\Delta \vdash p_i : E_i$ holds for some Δ. The premises consist of finite (possibly empty) lists of more atomic formulae $(r_j\,p_j,\ldots,r_k\,p_k)$ and side-conditions $(c_1,\ldots,c_m)$. The intended

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

meaning of such a rule is that if all of the formulae and side-conditions in the premises are satisfied by a given instantiation of their variables, then one may deduce that the conclusion is satisfied under that same variable instantiation.

It is not immediately obvious what kinds of constraint c would give a useful model of inductive definitions which occur in practice. It seems that the absolute minimum is constraints of *name inequality* $x \neq x'$, with x and x' being variables of the same name sort N. The need for name inequality constraints arises from the fundamental asymmetry of pattern valuation – all occurrences of the same variable are *always* instantiated in the same way whereas different variables *may* also be instantiated the same, unless we explicitly state otherwise. In fact, name inequality constraints are necessary and sufficient to define full (α-)disequality for any nominal signature (Cheney & Urban, 2008).

In fact, we generalise from name inequality constraints to *freshness constraints* $x \# p$ between a variable x of name sort and a pattern p of any equality type. This follows standard practice from nominal logic (Pitts, 2003) and nominal logic programming (Cheney & Urban, 2008). The semantics of this freshness constraint is that the name x does not appear free in the term represented by p, in the sense of Definition 2.5. This will be formalised in Definition 3.7. In the case where $\Delta \vdash p : N$, the freshness constraint actually reduces to a name inequality constraint. For convenience we also include *equality constraints*, which we interpret as α-equivalence constraints on the underlying set of ground trees or, equivalently, as simple equality on the corresponding α-trees.

Definition 3.3 (*Atomic constraints and schematic formulae*)
The set $Constr_\Sigma$ of atomic constraints is defined by the following grammar:

$$c \in Constr_\Sigma ::= p = p \mid x \# p$$

These are used to build up the set of schematic formulae $Form_\Sigma$, which is defined by the following grammar:

$$\varphi \in Form_\Sigma ::= r_i\, p \mid c \mid \mathsf{T} \mid \mathsf{F} \mid \varphi \,\&\, \varphi \mid \varphi \vee \varphi \mid \exists x : E.\, \varphi$$

This grammar contains atomic formulae and constraints, constants denoting true and false, conjunction, disjunction and existential quantification respectively. Here r_i is a member of the fixed, finite set of relation symbols $\{r_1, \ldots, r_n\}$. The only meta-level binder is in the existential formula, where x is bound in the formula φ.

Figure 7 presents rules defining a well-formedness judgement $\Delta \vdash \varphi\ ok$ for schematic formulae. The rules are standard – the case for an atomic formula assumes that the relation symbols $r_1, \ldots, r_n$ are associated with equality types $E_1, \ldots, E_n$ respectively. The side condition for the existential rule requires α-conversion at the meta-level in order to satisfy the side-condition that x be a fresh variable, and the rule for freshness constraints requires that the pattern on the left-hand side of the # be assigned a name sort N – as discussed above, this can only be satisfied if p is actually a variable x such that $\Delta(x) = N$.

We can now use schematic formulae to define inductive rules, of the form introduced above, and complete α-inductive definitions composed of these. Since the grammar of formulae includes atomic constraints and conjunctions, it suffices to consider schematic rules whose premise is a single formula φ. This corresponds to the typical presentation of inductively defined relations in structural operational semantics.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$\frac{\Delta \vdash p : E \qquad \Delta \vdash p' : E}{\Delta \vdash p = p'\ ok} \qquad \frac{\Delta \vdash x : N \qquad \Delta \vdash p : E}{\Delta \vdash x \# p\ ok} \qquad \frac{\psi \in \{\mathsf{T}, \mathsf{F}\}}{\Delta \vdash \psi\ ok} \qquad \frac{r_i \subseteq E_i \qquad \Delta \vdash p : E_i}{\Delta \vdash r_i\, p\ ok}$$

$$\frac{\Delta \vdash \varphi\ ok \qquad \Delta \vdash \varphi'\ ok}{\Delta \vdash \varphi\ \&\ \varphi'\ ok} \qquad \frac{\Delta \vdash \varphi\ ok \qquad \Delta \vdash \varphi'\ ok}{\Delta \vdash \varphi \vee \varphi'\ ok} \qquad \frac{x \notin dom(\Delta) \qquad \Delta, x : E \vdash \varphi\ ok}{\Delta \vdash \exists x : E.\, \varphi\ ok}$$

Fig. 7. Typing rules for schematic constraints and formulae.

Definition 3.4 (*Schematic rules*)
A *schematic rule* has the form

$$\frac{\varphi}{r_i\, p} \tag{1}$$

where $i \in \{1, \ldots, n\}$. The rule in Equation (1) is *well-formed* if there exists a type environment Δ such that $dom(\Delta) = vars(p)$, for which $\Delta \vdash p : E_i$ and $\Delta \vdash \varphi\ ok$ both hold. It is not hard to see that if such an environment Δ exists then it is unique. Thus, any variables that appear in the premise but not in the conclusion must be existentially quantified in the premise.

Definition 3.5 (α-*inductive definitions*)
An α-*inductive definition* $\mathscr{D}$ is a finite set of well-formed schematic rules.

3.3 Semantics of α-inductive definitions

Before we address the semantics of α-inductive definitions, we will present some transformations on schematic rules and definitions that will greatly simplify the presentation and the proofs. We have already seen one such simplification in the presentation of α-inductive definitions – in Definition 3.4 we argued that it suffices to only consider schematic rules with a single formula φ on the top line.

At the expense of extending the nominal signature we will consider only α-inductive definitions where the rules use just a single relation symbol – we will fix the relation symbol r for this purpose. Suppose we have a nominal signature Σ and an α-inductive definition $\mathscr{D}$ (as defined in Definition 3.5) concerning relation symbols $r_1, \ldots, r_n$, which represent subsets of the equality types $E_1, \ldots, E_n$ respectively. We extend Σ to produce a new signature Σ', which is related to the original signature as follows:

- $\mathbb{N}_{\Sigma'} \equiv \mathbb{N}_\Sigma$
- $\mathbb{S}_{\Sigma'} \equiv \mathbb{S}_\Sigma \uplus \{S_r\}$
- $\mathbb{C}_{\Sigma'} \equiv \mathbb{C}_\Sigma \uplus \{R_1 : E_1 \to S_r, \ldots, R_n : E_n \to S_r\}$,

where S_r is a new nominal data sort which is used to represent inductively defined relations, which must not appear either in $\mathbb{S}_\Sigma$ or $\mathbb{N}_\Sigma$. We write $r \subseteq S_r$ to indicate this. We then represent the original relation symbols $r_1, \ldots, r_n$ using n new, distinct constructors $R_1, \ldots, R_n$, whose argument types correspond to the types of the original relations. The schematic rules are altered by replacing every atomic formula $r_i\, p$ by the atomic formula $r(R_i\, p)$, and the task of matching against the relation symbol in an atomic formula is now done by solving equality constraints on schematic patterns. We exploit the fact that subsets of $\alpha\text{-}Tree_{\Sigma'}(S_r)$ are in bijection with n-tuples of subsets of $\alpha\text{-}Tree_\Sigma(E_1), \ldots, \alpha\text{-}Tree_\Sigma(E_n)$. The intended

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

meaning of the original α-inductive definition is preserved, although we do not define this formally. Henceforth, when dealing with α-inductive definitions (or their encodings) we will assume that this construction has already been applied to the nominal signature Σ.

Our final simplification allows us to reduce all α-inductive definitions (now only involving a single relation symbol) to only use a single inductive rule (Clark, 1978). This is possible because of the presence of equality constraints, disjunction and existential quantification in our grammar of formulae. To illustrate this translation, suppose that we have a definition $\mathscr{D}$ (as defined in Definition 3.5) which uses a single relation symbol r but has n rules:

$$\frac{\varphi_1}{r\,p_1} \qquad \frac{\varphi_2}{r\,p_2} \qquad \cdots \qquad \frac{\varphi_{n-1}}{r\,p_{n-1}} \qquad \frac{\varphi_n}{r\,p_n}$$

The presence of multiple rules is an implicit disjunction, which suggests that we may be able to combine the premises of the rules using an n-way disjunction, provided that we can construct a single conclusion for the combined rule. To do this, we fix a new variable x which does not occur in the rules for $\mathscr{D}$ presented above, and use the atomic formula $r\,x$ as the conclusion of the combined rule. The variable x stands for the ground predicate instance for which we are trying to construct a derivation. To construct the corresponding premise (for the ith rule above) we existentially quantify the variables which occur in p_i and require that x is α-equivalent to p_i. We can then process the original premise φ_i as normal. The rules for $\mathscr{D}$ presented above therefore become the single rule,

$$\frac{(\exists vars(p_1).\, x = p_1 \;\&\; \varphi_1) \vee \cdots \vee (\exists vars(p_n).\, x = p_n \;\&\; \varphi_n)}{r\,x} \tag{2}$$

where we write $\exists vars(p).\,\varphi$ for the iterated $\exists$-quantification of all of the variables appearing in p (with the corresponding type annotations for the variables in $vars(p)$). Rule (2) constitutes a new α-inductive definition whose semantics is identical to that of $\mathscr{D}$. This is a straightforward consequence of the semantics of schematic formulae from Definition 3.9, so we omit the proof. Henceforth, we will only consider α-inductive definitions with a single inference rule and a single relation symbol, which we say are in *standard form*.

Definition 3.6 (*α-inductive definitions in standard form*)
An α-inductive definition $\mathscr{D}$ of a set of α-trees of equality type S_r is in *standard form* if it consists of a single inference rule

$$\frac{\varphi}{r\,x} \tag{3}$$

where $\{x : S_r\} \vdash \varphi\ ok$ holds (i.e. x is the only free variable in φ), $r \subseteq S_r$ and r is the only relation symbol that appears in φ.

We now work towards a semantics for α-inductive definitions in standard form, and the first step in this direction is to define satisfaction of atomic constraints. It is straightforward to show that the relation symbol r may not appear in well-formed patterns, and hence will never appear in well-formed atomic constraints. Hence, the constraint satisfaction relation has the form $V \models c$, since we need a valuation V to instatiate any variable occurring in c, but do not need to consider the semantics of relation r.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$\frac{[\![p]\!]_V \in \mathfrak{R}}{(\mathfrak{R},V) \models r\,p} \qquad \frac{V \models c}{(\mathfrak{R},V) \models c} \qquad \frac{(\mathfrak{R},V) \models \varphi_1 \qquad (\mathfrak{R},V) \models \varphi_2}{(\mathfrak{R},V) \models \varphi_1 \,\&\, \varphi_2}$$

$$\frac{}{(\mathfrak{R},V) \models \mathsf{T}} \qquad \frac{(\mathfrak{R},V) \models \varphi_1}{(\mathfrak{R},V) \models \varphi_1 \vee \varphi_2} \qquad \frac{(\mathfrak{R},V) \models \varphi_2}{(\mathfrak{R},V) \models \varphi_1 \vee \varphi_2}$$

$$\frac{x \notin dom(V) \qquad t \in \alpha\text{-}Tree_{\Sigma}(E) \qquad (\mathfrak{R},V[x \mapsto t]) \models \varphi}{(\mathfrak{R},V) \models \exists x : E.\,\varphi}$$

Fig. 8. Formula satisfaction rules.

Definition 3.7 (*Satisfaction of atomic constraints*)
If $\Delta \vdash c\ ok$ and $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ then we define satisfaction of atomic constraints by cases, as follows:

$$\begin{aligned} V \models p = p' &\iff [\![p]\!]_V = [\![p']\!]_V \\ V \models x \# p &\iff V(x) \not\approx [\![p]\!]_V . \end{aligned}$$

The simplicity of this definition stems from the fact that patterns denote α-equivalence classes, so α-equivalence is handled implicitly in the semantics. We now consider the semantics of schematic formulae arising from α-inductive definitions in standard form. As for atomic constraints, the satisfaction judgement must involve a valuation V with the appropriate domain. However, because the relation symbol r may appear in formulae, we need to interpret it using an α-tree relation, which is just the ground term model in α-trees for the relation.

Definition 3.8 (α-*tree relations*)
An α-*tree relation* $\mathfrak{R}$ over the nominal signature Σ is a subset $\mathfrak{R} \subseteq \alpha\text{-}Tree_{\Sigma}(S_r)$, where $S_r \in \mathbb{S}_{\Sigma}$ is the nominal data sort such that $r \subseteq S_r$, i.e. which contains terms of the form $r\,p$.

Definition 3.9 (*Satisfaction of formulae*)
Suppose that $r \subseteq S_r$, $\Delta \vdash \varphi\ ok$, $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ and $\mathfrak{R} \subseteq \alpha\text{-}Tree_{\Sigma}(S_r)$ all hold, with S_r and $\mathfrak{R}$ defined as in Definition 3.8. Then satisfaction of formulae is written $(\mathfrak{R},V) \models \varphi$ and is defined by the rules in Figure 8. We write $V[x \mapsto t]$ for the valuation with domain $dom(V) \uplus \{x\}$ (where $x \notin dom(V)$) that maps x to t and otherwise behaves like V.

Again, α-equivalence is handled implicitly but formally. By using variables that range over α-equivalence classes directly we build in α-equivalence from the ground up. Most of these rules are completely standard, which is one of the advantages of our approach. Note that the rule for existential formulae $\exists x : E.\,\varphi$ has the hypothesis that there must exist a ground tree $t \in \alpha\text{-}Tree_{\Sigma}(E)$. Therefore, the judgement $(\mathfrak{R},V) \models \exists x : E.\,\varphi$ cannot hold if E is an empty equality type. The satisfaction rule for atomic formulae simply requires that the α-tree $[\![p]\!]_V$ produced by instantiating the pattern p with the valuation V is a member of the α-tree relation $\mathfrak{R}$. We can now present a semantics for α-inductive definitions in standard form.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$\frac{\mathsf{T}}{\mathtt{sub}(\mathtt{Var}\,z,z,t',t')} \qquad \frac{z\#y}{\mathtt{sub}(\mathtt{Var}\,y,z,t',\mathtt{Var}\,y)} \qquad \frac{\mathsf{T}}{\mathtt{sub}(\mathtt{Lam}{<}z{>}t,z,t',\mathtt{Lam}{<}z{>}t)}$$

$$\frac{y\#(z,t')\ \&\ \mathtt{sub}(t,z,t',t'')}{\mathtt{sub}(\mathtt{Lam}{<}y{>}t,z,t',\mathtt{Lam}{<}y{>}t'')} \qquad \frac{\mathtt{sub}(t_1,z,t',t_1')\ \&\ \mathtt{sub}(t_2,z,t',t_2')}{\mathtt{sub}(\mathtt{App}(t_1,t_2),z,t',\mathtt{App}(t_1',t_2'))}$$

$$\frac{\mathtt{sub}(t,z,t',t'')}{\mathtt{beta}(\mathtt{App}((\mathtt{Lam}{<}z{>}t),t'),t'')} \qquad \frac{\mathtt{beta}(t,t'')}{\mathtt{beta}(\mathtt{App}(t,t'),\mathtt{App}(t'',t'))}$$

$$\frac{\mathsf{T}}{\mathtt{nf}(\mathtt{Var}\,z)} \qquad \frac{\mathsf{T}}{\mathtt{nf}(\mathtt{Lam}{<}z{>}t)} \qquad \frac{\mathsf{T}}{\mathtt{nf}(\mathtt{App}(\mathtt{Var}\,z,t))} \qquad \frac{\mathtt{nf}(\mathtt{App}(t_1,t_2))}{\mathtt{nf}(\mathtt{App}(\mathtt{App}(t_1,t_2),t_3))}$$

$$\frac{\mathtt{nf}(t)}{\mathtt{betas}(t,t)} \qquad \frac{\mathtt{beta}(t,t'')\ \&\ \mathtt{betas}(t'',t')}{\mathtt{betas}(t,t')}$$

Fig. 9. Schematic rules for β-reduction of untyped λ-terms from Example 3.11.

Definition 3.10 (*Semantics of α-inductive definitions in standard form*)
The *denotation* $[\![\mathscr{D}]\!] \subseteq \alpha\text{-}Tree_{\Sigma}(S_r)$ of a standard α-inductive definition $\mathscr{D}$ (as in Definition 3) is the least fixed point of the function $\Phi_{\mathscr{D}}$ on subsets of α-trees defined by

$$\Phi_{\mathscr{D}}(\mathfrak{R}) \equiv \{t \in \alpha\text{-}Tree_{\Sigma}(S_r) \mid (\mathfrak{R}, \{x \mapsto t\}) \models \varphi\}. \tag{4}$$

The notation $\{x \mapsto t\}$ represents the valuation V which has $dom(V) = \{x\}$ and maps x to the α-tree t. The least fixed point exists by Tarski's fixed point theorem (Tarski, 1955), since $\Phi_{\mathscr{D}}$ is monotone in the sense that if $\mathfrak{R} \subseteq \mathfrak{R}' \subseteq \alpha\text{-}Tree_{\Sigma}(S_r)$ and $(\mathfrak{R}, V) \models \varphi$ then $(\mathfrak{R}', V) \models \varphi$.

Furthermore, since $\Phi_{\mathscr{D}}$ is finitary we can construct $[\![\mathscr{D}]\!]$ as the union of a chain of subsets of $\alpha\text{-}Tree_{\Sigma}(S_r)$, as illustrated by the following Lemma.

Lemma 3.1 (*Compactness of denotations of α-inductive definitions*)
For any α-inductive definition $\mathscr{D}$ in standard form, we can construct $[\![\mathscr{D}]\!] = \bigcup_{n\in\mathbb{N}} [\![\mathscr{D}]\!]^{(n)}$, where $[\![\mathscr{D}]\!]^{(n)}$ is the n-fold application $\Phi^n_{\mathscr{D}}(\emptyset)$.

Example 3.11 (*β-reduction as an α-inductive definition*)
Recalling once again the nominal signature Λ for untyped λ-terms from Example 2.2, we will define β-reduction as a collection of schematic rules of the form presented above. The rules will carve out relations $\mathtt{beta} \subseteq \mathtt{term} * \mathtt{term}$ denoting single-step call-by-name β-reduction and $\mathtt{betas} \subseteq \mathtt{term} * \mathtt{term}$ denoting maximal sequences of reductions using $\mathtt{beta}$. These definitions require an auxiliary relation $\mathtt{sub} \subseteq \mathtt{term} * \mathtt{var} * \mathtt{term} * \mathtt{term}$ which corresponds to capture-avoiding substitution in the sense that $\mathtt{sub}(t,z,t',t'')$ holds iff $t[t'/z] =_{\alpha} t''$, and a predicate $\mathtt{nf} \subseteq \mathtt{term}$ which holds for terms in normal form. Schematic rules defining these relations are presented in Figure 9. It is not hard to show that $\mathtt{nf}(t)$ holds iff there exists no t' such that $\mathtt{beta}(t,t')$ holds.

Now, in order to convert this definition into standard form in the sense of Definition 3.6, we extend Λ to Λ' by adding a new nominal data sort S_r and four new constructors ($\mathtt{sub}$,

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

$$\frac{\begin{array}{l}(\exists z : \mathtt{var}.\, \exists t' : \mathtt{term}.\, x = \mathtt{sub}\,(\mathtt{Var}\,z, z, t', t') \;\&\; \mathsf{T}) \;\vee \\ (\exists y, z : \mathtt{var}.\, \exists t' : \mathtt{term}.\, x = \mathtt{sub}\,(\mathtt{Var}\,y, z, t', \mathtt{Var}\,y) \;\&\; z \# y) \;\vee \\ (\exists z : \mathtt{var}.\, \exists t, t' : \mathtt{term}.\, x = \mathtt{sub}\,(\mathtt{Lam}\!<\!z\!>\!t, z, t', \mathtt{Lam}\!<\!z\!>\!t) \;\&\; \mathsf{T}) \;\vee \\ (\exists y, z : \mathtt{var}.\, \exists t, t', t'' : \mathtt{term}.\, x = \mathtt{sub}\,(\mathtt{Lam}\!<\!y\!>\!t, z, t', \mathtt{Lam}\!<\!y\!>\!t'') \;\& \\ \quad y \# (z, t') \;\&\; r\,(\mathtt{sub}\,(t, z, t', t''))) \;\vee \\ (\exists z : \mathtt{var}.\, \exists t_1, t_1', t_2, t_2', t' : \mathtt{term}.\, x = \mathtt{sub}\,(\mathtt{App}\,(t_1, t_2), z, t', \mathtt{App}\,(t_1', t_2')) \;\& \\ \quad r\,(\mathtt{sub}\,(t_1, z, t', t_1')) \;\&\; r\,(\mathtt{sub}\,(t_2, z, t', t_2'))) \;\vee \\ (\exists z : \mathtt{var}.\, \exists t, t', t'' : \mathtt{term}.\, x = \mathtt{beta}\,(\mathtt{App}\,((\mathtt{Lam}\!<\!z\!>\!t), t'), t'') \;\&\; r\,(\mathtt{sub}\,(t, z, t', t''))) \;\vee \\ (\exists t, t', t'' : \mathtt{term}.\, x = \mathtt{beta}\,(\mathtt{App}\,(t, t'), \mathtt{App}\,(t'', t')) \;\&\; r\,(\mathtt{beta}\,(t, t''))) \;\vee \\ (\exists z : \mathtt{var}.\, x = \mathtt{nf}\,(\mathtt{Var}\,z) \;\&\; \mathsf{T}) \;\vee \\ (\exists z : \mathtt{var}.\, \exists t : \mathtt{term}.\, x = \mathtt{nf}\,(\mathtt{Lam}\!<\!z\!>\!t) \;\&\; \mathsf{T}) \;\vee \\ (\exists z : \mathtt{var}.\, \exists t : \mathtt{term}.\, x = \mathtt{nf}\,(\mathtt{App}\,(z, t)) \;\&\; \mathsf{T}) \;\vee \\ (\exists t_1, t_2, t_3 : \mathtt{term}.\, x = \mathtt{nf}\,(\mathtt{App}\,(\mathtt{App}\,(t_1, t_2), t_3)) \;\&\; r\,(\mathtt{nf}\,(\mathtt{App}\,(t_1, t_2)))) \;\vee \\ (\exists t : \mathtt{term}.\, x = \mathtt{betas}\,(t, t) \;\&\; r\,(\mathtt{nf}\,(t))) \;\vee \\ (\exists t, t', t'' : \mathtt{term}.\, x = \mathtt{betas}\,(t, t') \;\&\; r\,(\mathtt{beta}\,(t, t'')) \;\&\; r\,(\mathtt{betas}\,(t'', t')))\end{array}}{r\,x}$$

Fig. 10. β-reduction of untyped λ-terms from Example 3.11 presented as an α-inductive definition in standard form.

$\mathtt{beta}$, $\mathtt{nf}$ and $\mathtt{betas}$) so that $\mathbb{N}_{\Lambda'} \equiv \{\mathtt{var}\}$, $\mathbb{S}_{\Lambda'} \equiv \{\mathtt{term}, S_r\}$ and

$$\begin{array}{rcl}\mathbb{C}_{\Lambda'} & \equiv & \{\mathtt{Var} : \mathtt{var} \to \mathtt{term},\ \mathtt{App} : \mathtt{term} * \mathtt{term} \to \mathtt{term},\ \mathtt{Lam} : [\mathtt{var}]\mathtt{term} \to \mathtt{term}, \\ & & \mathtt{sub} : \mathtt{term} * \mathtt{var} * \mathtt{term} * \mathtt{term} \to S_r,\ \mathtt{beta} : \mathtt{term} * \mathtt{term} \to S_r, \\ & & \mathtt{nf} : \mathtt{term} \to S_r,\ \mathtt{betas} : \mathtt{term} * \mathtt{term} \to S_r\}.\end{array}$$

Then the collection of schematic rules over Λ presented above can be translated into the (somewhat unwieldy) α-inductive definition $\mathscr{L}$ over Λ' presented in Figure 10. That definition comprises a single rule with a single relation symbol (r) and a single free variable (x), and is hence in standard form.

3.4 α-inductive definitions and equivariance

We end this section with a brief discussion of the relationship between α-inductive definitions and the concept of equivariance from nominal logic. Name-permutations are a staple of most nominal techniques for abstract syntax involving binders (Gabbay & Pitts, 2002; Pitts, 2003; Cheney & Urban, 2008), which we have avoided thus far because α-equivalence is handled implicitly by the direct use of α-equivalence classes of ground trees to produce the term-model semantics for α-inductive definitions.

Definition 3.12 (*Permutations*)
Permutations $\pi \in \mathit{Perm}$ are bijections from Name to Name which are *finite* (i.e. the set $\{n \in \mathit{Name} \mid \pi(n) \neq n\}$ is finite) and *sort-respecting* (i.e. $\mathit{sort}(\pi(n)) = \mathit{sort}(n)$ for all $n \in \mathit{Name}$).

We define the *action* $\pi \cdot g \in \mathit{Tree}_\Sigma(E)$ of a permutation π on a ground tree $g \in \mathit{Tree}_\Sigma(E)$ as the ground tree that results from permuting all names occurring in g according to π (including those in abstraction position). Since this process respects α-equivalence, we get

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

a well-defined action on α-trees such that $\pi \cdot [g]_\alpha = [\pi \cdot g]_\alpha$. It follows that, equipped with this permutation action, the set of ground trees $Tree_\Sigma$ is a nominal set (Pitts, 2003).

Definition 3.13 (*Equivariant α-tree relations*)
An α-tree relation $\mathfrak{R}$ is *equivariant* if $\mathfrak{R} \subseteq \pi \cdot \mathfrak{R}$ for all π, where $\pi \cdot \mathfrak{R} \equiv \{\pi \cdot t \mid t \in \mathfrak{R}\}$.

Equivariance is a fundamental concept in nominal logic (Pitts, 2003). Informally, an equivariant α-tree relation is one that is closed under permutation of names, which means that the membership of α-equivalence class t in the relation is not dependent on the particular names that occur within the representatives of the α-equivalence class. We view this as a desirable property because the underlying representation of names and binding is an implementation detail that should be hidden from the programmer if at all possible.

Definition 3.14 (*Action of a permutation on a valuation*)
Given an α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and a permutation π, we write $\pi \cdot V$ for the α-tree valuation in $\alpha\text{-}Tree_\Sigma(\Delta)$ which maps x to $\pi \cdot t$ if $V(x) = t$.

Given these definitions, it is not hard to show that constraint and formula satisfaction are equivariant, i.e. that $V \models c \Longrightarrow \pi \cdot V \models c$ and $(\mathfrak{R}, V) \models \varphi \Longrightarrow (\pi \cdot \mathfrak{R}, \pi \cdot V) \models \varphi$ hold, respectively. Furthermore, we obtain the following important result.

Theorem 3.15 (*Equivariance of denotations of α-inductive definitions*)
The denotation $[\![\mathscr{D}]\!] \subseteq \alpha\text{-}Tree_\Sigma(S_r)$ of any α-inductive definition $\mathscr{D}$ is an equivariant α-tree relation.

This result proves that the abstraction boundary between the user's view (of names as meta-variables) and the internal view (where the denotation of an α-inductive definitions is a set of α-equivalence classes over a term language involving particular ground names) can never be breached. Intuitively, this is because users cannot write down α-inductive definitions whose meaning depends on particular ground names.

4 Encoding α-inductive definitions in αML

In this section we embed α-inductive definitions into the αML meta-language in a simple and convenient way. We then relate the operational semantics of αML to that of a simple CLP language over the constraint domain of α-trees with equality and freshness constraints, and prove soundness and completeness results for the operational behaviour of embedded α-inductive definitions in αML.

As in Section 3.3, we will fix a single relation symbol r, which we will regard as an αML variable of type $S_r \to \mathtt{prop}$, for some fixed nominal data sort S_r. We assume that $\mathscr{D}$ is an α-inductive definition of the form specified in Definition 3.6. We will identify $\mathscr{D}$ with a closed αML recursive function value $v_\mathscr{D}$ of type $S_r \to \mathtt{prop}$, as follows:

$$v_\mathscr{D} \equiv (\mathtt{fun}\, r(x : S_r) : \mathtt{prop} = \lceil \varphi \rceil) \qquad \text{where } \mathscr{D} \equiv \frac{\varphi}{r\,x}$$

Hence, we must translate the formula $\varphi \in Form_\Sigma$ into an αML expression $\lceil \varphi \rceil$ such that $\{r : S_r \to \mathtt{prop}, x : S_r\} \vdash \lceil \varphi \rceil : \mathtt{prop}$ holds. A translation function $\lceil - \rceil$ which satisfies this

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

requirement is defined by the following rules:

$$\lceil \mathsf{T} \rceil \equiv \mathsf{T} \qquad \lceil r\,p \rceil \equiv r\,p \qquad \lceil c \rceil \equiv c \qquad \lceil \exists x : E.\,\varphi \rceil \equiv \exists x : E.\,\lceil \varphi \rceil$$

$$\lceil \varphi_1 \vee \varphi_2 \rceil \equiv \lceil \varphi_1 \rceil \parallel \lceil \varphi_2 \rceil \qquad \lceil \varphi_1 \;\&\; \varphi_2 \rceil \equiv \lceil \varphi_1 \rceil \;\&\; \lceil \varphi_2 \rceil$$

$$\lceil \mathsf{F} \rceil \equiv \exists x : N.\, x \# x \;\; (\text{for some } N \in \mathbb{N}_\Sigma)$$

where e_1 & e_2 are defined as syntactic sugar in αML for the sequential execution of e_1 followed by e_2, which can be implemented as $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$, for some $x \notin FV(e_2)$.

Schematic patterns of type E correspond exactly to αML values of the same equality type, so atomic constraints and atomic formulae are translated unchanged. The intuition is that the atomic formula $r\,p$ corresponds to passing the value p into the recursive function $v_{\mathscr{D}}$ corresponding to $\mathscr{D}$, which will be substituted for r. The T formula is also translated directly, and the existential and disjunction formulae are translated by a simple recursive step. The only formulae whose translation is non-trivial are the final two: conjunctions are implemented using `let` bindings and the eager reduction strategy of αML to get a left-to-right sequential conjunction, and the translation of a false formula generates a variable x of some name sort $N \in \mathbb{N}_\Sigma$ and then fails finitely because no name can be fresh for itself.

Thus, given an inductive definition $\mathscr{D}$ (with its associated αML function $v_{\mathscr{D}}$), every schematic formula $\varphi \in \mathit{Form}_\Sigma$ has a straightforward encoding as an αML expression $\lceil \varphi \rceil [v_{\mathscr{D}}/r]$, and it follows that the recursive function value $v_{\mathscr{D}}$ corresponding to $\mathscr{D}$ satisfies $\emptyset \vdash v_{\mathscr{D}} : S_r \to \texttt{prop}$. The syntax of schematic formulae and the αML meta-language were designed so that the translation of schematic formulae into αML expressions would be largely trivial. Hence, we overload the φ meta-variable to refer both to a schematic formula and its corresponding αML expression $\lceil \varphi \rceil$, since they are so similar.

Example 4.1 (β*-reduction as an* α*ML recursive function*)
Continuing the example of λ-calculus β-reduction from Example 3.11, we note that the α-inductive definition $\mathscr{L}$ presented in Figure 10 was in standard form. Hence, we can translate $\mathscr{L}$ into the following αML recursive function $v_{\mathscr{L}}$, for which $\emptyset \vdash v_{\mathscr{L}} : S_r \to \texttt{prop}$ holds.

$$\begin{aligned}
&\texttt{fun}\, r(x : S_r) : \texttt{prop} = ((\exists z : \texttt{var}.\, \exists t' : \texttt{term}.\, x = \texttt{sub}\,(\texttt{Var}\, z, z, t', t') \;\&\; \mathsf{T}) \\
&\quad \parallel (\exists y, z : \texttt{var}.\, \exists t' : \texttt{term}.\, x = \texttt{sub}\,(\texttt{Var}\, y, z, t', \texttt{Var}\, y) \;\&\; z \# y) \\
&\quad \parallel (\exists z : \texttt{var}.\, \exists t, t' : \texttt{term}.\, x = \texttt{sub}\,(\texttt{Lam}{<}z{>}t, z, t', \texttt{Lam}{<}z{>}t) \;\&\; \mathsf{T}) \\
&\quad \parallel (\exists y, z : \texttt{var}.\, \exists t, t', t'' : \texttt{term}.\, x = \texttt{sub}\,(\texttt{Lam}{<}y{>}t, z, t', \texttt{Lam}{<}y{>}t'') \;\& \\
&\qquad\; y \# (z, t') \;\&\; r\,(\texttt{sub}\,(t, z, t', t''))) \\
&\quad \parallel (\exists z : \texttt{var}.\, \exists t_1, t_1', t_2, t_2', t' : \texttt{term}.\, x = \texttt{sub}\,(\texttt{App}\,(t_1, t_2), z, t', \texttt{App}\,(t_1', t_2')) \;\& \\
&\qquad\; r\,(\texttt{sub}\,(t_1, z, t', t_1')) \;\&\; r\,(\texttt{sub}\,(t_2, z, t', t_2'))) \\
&\quad \parallel (\exists z : \texttt{var}.\, \exists t, t', t'' : \texttt{term}.\, x = \texttt{beta}\,(\texttt{App}\,((\texttt{Lam}{<}z{>}t), t'), t'') \;\&\; r\,(\texttt{sub}\,(t, z, t', t''))) \\
&\quad \parallel (\exists t, t', t'' : \texttt{term}.\, x = \texttt{beta}\,(\texttt{App}\,(t, t'), \texttt{App}\,(t'', t')) \;\&\; r\,(\texttt{beta}\,(t, t''))) \\
&\quad \parallel (\exists z : \texttt{var}.\, x = \texttt{nf}\,(\texttt{Var}\, z) \;\&\; \mathsf{T}) \\
&\quad \parallel (\exists z : \texttt{var}.\, \exists t : \texttt{term}.\, x = \texttt{nf}\,(\texttt{Lam}{<}z{>}t) \;\&\; \mathsf{T}) \\
&\quad \parallel (\exists z : \texttt{var}.\, \exists t : \texttt{term}.\, x = \texttt{nf}\,(\texttt{App}\,(z, t)) \;\&\; \mathsf{T}) \\
&\quad \parallel (\exists t_1, t_2, t_3 : \texttt{term}.\, x = \texttt{nf}\,(\texttt{App}\,(\texttt{App}\,(t_1, t_2), t_3)) \;\&\; r\,(\texttt{nf}\,(\texttt{App}\,(t_1, t_2)))) \\
&\quad \parallel (\exists t : \texttt{term}.\, x = \texttt{betas}\,(t, t) \;\&\; r\,(\texttt{nf}\,(t))) \\
&\quad \parallel (\exists t, t', t'' : \texttt{term}.\, x = \texttt{betas}\,(t, t') \;\&\; r\,(\texttt{beta}\,(t, t'')) \;\&\; r\,(\texttt{betas}\,(t'', t'))))
\end{aligned}$$

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

(F1)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};r\,p) \rightsquigarrow \exists\Delta(\overline{c};\vec{\varphi};\varphi[p/x])$	if $v_{\mathscr{D}} = \mathtt{fun}\, r(x:S_r):\mathtt{prop} = \varphi$.
(F2)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};c) \rightsquigarrow \exists\Delta(\overline{c}\ \&\ c;\vec{\varphi};\mathsf{T})$	if $\not\models \exists\Delta(\overline{c}\ \&\ c)$.
(F3)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi},\varphi;\mathsf{T}) \rightsquigarrow \exists\Delta(\overline{c};\vec{\varphi};\varphi)$	
(F4)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};\varphi\ \&\ \varphi') \rightsquigarrow \exists\Delta(\overline{c};\vec{\varphi},\varphi';\varphi)$	
(F5)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};\varphi_1 \vee \varphi_2) \rightsquigarrow \exists\Delta(\overline{c};\vec{\varphi};\varphi_i)$	if $i \in \{1,2\}$.
(F6)	$\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};\exists x:E.\,\varphi) \rightsquigarrow \exists\Delta,x:E(\overline{c};\vec{\varphi};\varphi)$	if $x \notin dom(\Delta)$.

Fig. 11. Formula reduction.

Note that the use of translated schematic formulae as αML expressions gives rise to the logical conjunction syntax in the definition of this function, as described above.

4.1 Constraint logic programming in αML

Figure 11 defines a transition relation $\mathscr{D} \vdash \exists\Delta(\overline{c};\vec{\varphi};\varphi) \rightsquigarrow \exists\Delta'(\overline{c}';\vec{\varphi}';\varphi')$ which we call *formula reduction*. The relation involves a special kind of configuration that contains a queue $\vec{\varphi}$ of formulae instead of a frame stack and where the expression at the top of the stack always corresponds to some formula φ. These are related to impure αML configurations as outlined below, and may be interpreted as goal states in a CLP language. The judgement is moderated by an α-inductive definition $\mathscr{D}$ because in order to evaluate an atomic formula $r\,p$ we must know the interpretation of the relation r. These rules give a largely standard formulation of the operational semantics of CLP as presented in Section 3 of Jaffar *et al.* (1998).

The formula reduction rules bear a striking similarity to certain rules from the operational semantics of αML presented in Figure 3. Rule (F1) is a specialised version of the application rule (P3). The constraint rule (F2) is identical to the impure rule (I2), and rule (F3), which moves on from a true formula to process the next formula in the queue, corresponds to rule (P1) in the case where the value is of type `prop`. Rule (F4) deals with conjunction by emulating the rules that deal with `let` bindings, as this is how conjunction is encoded within αML, and rules (P5) and (P6) are identical to the impure rules for handling branching and existential quantification, respectively. Thus, it is straightforward to relate formula reduction to the impure αML reduction rules from Figure 3.

Definition 4.2 (*CLP goal lists in αML*)
We encode a goal list $\vec{\varphi}$ as an αML frame stack $F_{\vec{\varphi}}$, which we define by recursion on the length of the goal list, as follows:

$$F_{\emptyset} \equiv \mathtt{Id} \qquad\qquad F_{\vec{\varphi},\varphi} \equiv F_{\vec{\varphi}} \circ (x.\,\varphi)\ \ (\text{for some } x \notin FV(\varphi)).$$

A frame stack corresponding to a CLP goal list is just a queue of formulae waiting to be evaluated. By type preservation, any formula that is successfully processed will result in the value T, and because the bound variable in each stack frame must not be free in the corresponding formula, this value is discarded at each step. The only information passed along is the constraints and the environment of generated variables.

The following result shows that formula reduction corresponds to a subset of the αML reduction rules. Thus, the operational semantics of αML incorporates both a standard functional programming language *and* a CLP language over the constraint domain of

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

α-trees. The proof proceeds by cases according to the structure of the formula φ, by matching the formula reduction rules with the corresponding impure reduction rules.

Theorem 4.3 (*Embedded CLP language*)
Let $\mathscr{D}$ be an α-inductive definition, and suppose that $\emptyset \vdash \exists\Delta(\overline{c}; F_{\vec{\varphi}}[v_{\mathscr{D}}/r]; \varphi[v_{\mathscr{D}}/r]) : \mathtt{prop}$ holds. Then an impure reduction $\exists\Delta(\overline{c}; F_{\vec{\varphi}}[v_{\mathscr{D}}/r]; \varphi[v_{\mathscr{D}}/r]) \longrightarrow \exists\Delta'(\overline{c}'; F; e)$ holds iff there exists a formula reduction $\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi')$ for some $\vec{\varphi}'$ and φ', with $F = F_{\vec{\varphi}'}[v_{\mathscr{D}}/r]$ and $e = \varphi'[v_{\mathscr{D}}/r]$.

Our aim in this section is to relate the evaluation of formulae in the αML operational semantics (actually, the formula reduction semantics) to the satisfaction of formulae as defined in Definition 3.9. We will use the following definition of the *set of solutions* to a configuration in the $\rightsquigarrow$ relation.

Definition 4.4 (*Solution sets*)
Let $\mathscr{D}$ be an inductively defined relation which we identify with $v_{\mathscr{D}} \equiv \mathtt{fun}\, r(x : S_r) : \mathtt{prop} = \varphi$. Given $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $\Delta \vdash c\ ok$ for all $c \in \overline{c}$ and $\Delta \vdash \psi\ ok$ for all $\psi \in \vec{\varphi}, \varphi$, we will write $solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ for the *solution set*

$$solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi) \;\equiv\; \{\exists\Delta'(\overline{c}') \mid \mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists\Delta, \Delta'(\overline{c}'; \emptyset; \mathsf{T}) \wedge \models \exists\Delta, \Delta'(\overline{c}')\}.$$

4.2 Logical soundness

Our first step towards demonstrating the correctness of formula reduction in αML is to prove a *logical soundness* result. This states that if a particular constraint problem is in the solution set of a configuration under the αML, then any valuation which satisfies the constraint problem also satisfies all of the formulae in the original configuration. Intuitively, this means that the αML operational semantics does not compute any wrong answers.

Definition 4.5 (*Formula entailment and equivalence*)
Given formulae φ and φ' such that $\Delta \vdash \varphi\ ok$ and $\Delta \vdash \varphi'\ ok$, we write $\mathscr{D}, \mathscr{D}' \models \forall\Delta.\, \varphi \Longrightarrow \varphi'$ to mean that, for all $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$, if $([\![\mathscr{D}]\!], V) \models \varphi$ then $([\![\mathscr{D}']\!], V) \models \varphi'$. We write $\mathscr{D}, \mathscr{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$ for the symmetric version of this relation. If $\mathscr{D} = \mathscr{D}'$ we abbreviate these to $\mathscr{D} \models \forall\Delta.\, \varphi \Longrightarrow \varphi'$ and $\mathscr{D} \models \forall\Delta.\, \varphi \equiv \varphi'$ respectively.

This notion of entailment between formulae will be used in our proof of logical soundness. We begin by enumerating some straightforward properties of pattern valuation with regard to weakening and substitution – see Lakin (2010)) for a proof outline for Lemma 4.2.

Lemma 4.1 (*Weakening properties of pattern valuation*)
Suppose that $\Delta \subseteq \Delta'$ and $V' \in \alpha\text{-}Tree_{\Sigma}(\Delta')$, and that $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ is the restriction of V' to $dom(\Delta)$. Then:

1. if $\Delta \vdash p : E$ then ($\Delta' \vdash p : E$ and) $[\![p]\!]_V = [\![p]\!]_{V'} \in \alpha\text{-}Tree_{\Sigma}(E)$.
2. if $\Delta \vdash c : \mathtt{prop}$ then ($\Delta' \vdash c : \mathtt{prop}$ and) $V \models c \Longleftrightarrow V' \models c$.
3. if $\Delta, \{r : S_r \to \mathtt{prop}\} \vdash \varphi : \mathtt{prop}$ and $\mathfrak{R} \subseteq \alpha\text{-}Tree_{\Sigma}(S_r)$ then ($\Delta', \{r : S_r \to \mathtt{prop}\} \vdash \varphi : \mathtt{prop}$ and) $(\mathfrak{R}, V) \models \varphi \Longleftrightarrow (\mathfrak{R}, V') \models \varphi$.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Lemma 4.2 (*Substitution properties of pattern valuation*)
Suppose that $\Delta, \{r : S_r \to \mathtt{prop}, x : E\} \vdash \varphi : \mathtt{prop}$ and $\Delta, \{x : E\} \vdash p' : E'$ and $\Delta \vdash p : E$ all hold. Then for any α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ and any α-tree relation $\mathfrak{R} \subseteq \alpha\text{-}Tree_\Sigma(S_r)$:

1. $[\![p'[p/x]]\!]_V = [\![p']\!]_{V[x \mapsto [\![p]\!]_V]}$.
2. $(\mathfrak{R}, V) \models \varphi[p/x] \iff (\mathfrak{R}, V[x \mapsto [\![p]\!]_V]) \models \varphi$.

The following is the main intermediate lemma needed to prove the logical soundness result. We prove that if the configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$ transitions to $\exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi')$, then any valuation that satisfies $(\overline{c}' \,\&\, \vec{\varphi}' \,\&\, \varphi')$ will also satisfy $(\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi)$. This means that the $\rightsquigarrow$ transition relation may narrow down the set of valuations that satisfy the configuration but it may not add extra satisfying valuations – it would be unsound to report satisfying valuations which do not satisfy the initial configuration. Here, and below, we interpret goal lists $\vec{\varphi}$ as implicit conjunctions of individual formulae so that they may be treated the same as normal schematic formulae. This is reasonable because the intended meaning of the goal list is that all formulae $\varphi \in \vec{\varphi}$ must be simultaneously satisfied. Then we can directly combine this conjunction with other atomic constraints $\overline{c}$ (also interpreted as an implicit conjunction) and formulae φ to produce a schematic formula $(\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi)$. Thus, we can then use our existing mathematical infrastructure to reason about the satisfaction of the entire configuration.

Lemma 4.3 (*Intermediate lemma for logical soundness*)
Let $\mathscr{D}$ be an α-inductive definition in the sense of Definition 4.4, and treat $\vec{\varphi}$ and $\overline{c}$ as implicit conjunctions as described above. Then if $\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi')$ holds, then $\Delta \subseteq \Delta'$ and $\mathscr{D} \models \forall\Delta'.\, (\overline{c}' \,\&\, \vec{\varphi}' \,\&\, \varphi') \Longrightarrow (\overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi)$.

Proof
We proceed by cases on the rule used to derive $\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta'(\overline{c}'; \vec{\varphi}'; \varphi')$.

(F1). In this case we have $\varphi = r\,p$, for some p. Using rule (F1) and the definition of $\mathscr{D}$ we get that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ and $\varphi' = \psi[p/x]$. Given some valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, \vec{\varphi} \,\&\, \psi[p/x]$, to prove the result it suffices to show that $([\![\mathscr{D}]\!], V) \models r\,p$ holds. By Lemma 4.2 we get that $([\![\mathscr{D}]\!], V[x \mapsto [\![p]\!]_V]) \models \psi$, and using Lemma 4.1 (and the definition of $[\![\mathscr{D}]\!]$ as the least fixed-point of a monotone operator in Definition 3.10) we get that $([\![\mathscr{D}]\!], V) \models r\,p$ holds, as required.

(F2). Here we have $\varphi = c$ for some c. Then it follows that $\Delta' = \Delta$, $\overline{c}' = \overline{c} \,\&\, c$, $\vec{\varphi}' = \vec{\varphi}$ and $\varphi' = \mathsf{T}$, and also that $\models \exists\Delta(\overline{c} \,\&\, c)$ holds. Given an arbitrary valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ such that $([\![\mathscr{D}]\!], V) \models (\overline{c} \,\&\, c) \,\&\, \vec{\varphi} \,\&\, \mathsf{T}$, it follows trivially that $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, \vec{\varphi} \,\&\, \varphi$ holds, as required.

(F3). In this case we know that $\vec{\varphi} = \vec{\varphi}^*, \varphi^*$ and $\varphi = \mathsf{T}$, for some $\vec{\varphi}^*$ and φ^*. We get that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}^*$ and $\varphi' = \varphi^*$ all hold. Then it is trivially the case that $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, \vec{\varphi}^* \,\&\, \varphi^*$ implies $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, (\vec{\varphi}^*, \varphi^*) \,\&\, \mathsf{T}$ holds for any $V \in \alpha\text{-}Tree_\Sigma(\Delta)$.

(F4). We get that $\varphi = \varphi_1 \,\&\, \varphi_2$, for some φ_1 and φ_2. By rule (F4) we have that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}, \varphi_2$ and $\varphi' = \varphi_1$. Then it follows that $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, (\vec{\varphi}, \varphi_2) \,\&\, \varphi_1$ implies $([\![\mathscr{D}]\!], V) \models \overline{c} \,\&\, \vec{\varphi} \,\&\, (\varphi_1 \,\&\, \varphi_2)$, as required.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

(F5). In this case, $\varphi = \varphi_1 \vee \varphi_2$ for some φ_1 and φ_2. Then we know that $\Delta' = \Delta$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ and either $\varphi' = \varphi_1$ or $\varphi' = \varphi_2$. In either of these cases we have $\mathscr{D} \models \forall \Delta.\, \varphi_j \Longrightarrow \varphi_1 \vee \varphi_2$, where $j \in \{1,2\}$. Thus, $([\![\mathscr{D}]\!], V) \models \overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi_j$ implies $([\![\mathscr{D}]\!], V) \models \overline{c}\ \&\ \vec{\varphi}\ \&\ (\varphi_1 \vee \varphi_2)$, for any $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$, as required.

(F6). We have that $\varphi = \exists x : E.\, \varphi^*$ for some φ^*, and by α-conversion we may assume that $x \notin dom(\Delta)$. Then by rule (F6) we get that $\Delta' = \Delta, x : E$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \varphi$ and $\varphi' = \varphi^*$. Given an arbitrary valuation $V \in \alpha\text{-}Tree_{\Sigma}(\Delta, x : E)$, we assume that $([\![\mathscr{D}]\!], V) \models \varphi^*$ holds. Writing V' for the restriction of V to $dom(\Delta)$, we get that $([\![\mathscr{D}]\!], V') \models \exists x : E.\, \varphi^*$, and then by Lemma 4.1 we have $([\![\mathscr{D}]\!], V) \models \exists x : E.\, \varphi^*$. Thus, we may conclude that $\mathscr{D} \models \forall \Delta, x : E.\, (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi^*) \Longrightarrow (\overline{c}\ \&\ \vec{\varphi}\ \&\ \exists x : E.\, \overline{c}^*)$, as required.

This completes the proof of Lemma 4.3. □

We are now in a position to prove the logical soundness lemma using Lemma 4.3.

Lemma 4.4 (*Logical soundness*)
With $\mathscr{D}$ and $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ as in Definition 4.4, for all $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ it is the case that if $\exists \Delta'(\overline{c}') \in solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ and $([\![\mathscr{D}]\!], V) \models \exists \Delta'(\overline{c}')$ then $([\![\mathscr{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$.

Proof
Let $\mathscr{D}$ be an α-inductive definition as in Definition 4.4. Given $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $\Delta \vdash c\ ok$ for all $c \in \overline{c}$ and $\Delta \vdash \psi\ ok$ for all $\psi \in \vec{\varphi}, \varphi$, and given an arbitrary α-tree valuation $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$, we assume that $\exists \Delta'(\overline{c}') \in solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ and $([\![\mathscr{D}]\!], V) \models \exists \Delta'(\overline{c}')$ both hold. It follows that there exist Δ' and $\overline{c}'$ such that $\mathscr{D} \vdash \exists \Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists \Delta, \Delta'(\overline{c}'; \emptyset; \mathsf{T})$ and $\models \exists \Delta, \Delta'(\overline{c}')$ both hold. Then by applying Lemma 4.3 to every individual $\rightsquigarrow$-transition in the above sequence, we get $\mathscr{D} \models \forall \Delta, \Delta'.\, \overline{c}' \Longrightarrow (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$, and since $([\![\mathscr{D}]\!], V) \models \exists \Delta'(\overline{c}')$, it follows that $([\![\mathscr{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$ holds, as required. □

4.3 Logical completeness

In this section we prove a *logical completeness* result, which is the converse of the logical soundness result from Lemma 4.4. This result states that if a configuration $\exists \Delta(\overline{c}; \vec{\varphi}; \varphi)$ is satisfied by a valuation V then there exists a computation path in the αML operational semantics which terminates at an element $\exists \Delta'(\overline{c}')$ of the solution set which is satisfied by V. Hence, αML does not discard any satisfying valuations during execution.

We first present a size metric on certain collections of satisfaction judgements, which we will use to show that formula reduction of satisfiable formulae eventually terminates. We begin by defining a size function $size(\varphi)$ on atomic formulae, such that $size(\varphi) \geqslant 1$ for all φ, as follows:

$$\begin{aligned} size(r\,p) = size(\mathsf{T}) &= 1 \\ size(c) &= 2 \\ size(\varphi_1\ \&\ \varphi_2) = size(\varphi_1 \vee \varphi_2) &= 1 + size(\varphi_1) + size(\varphi_2) \\ size(\exists x : E.\, \varphi) &= 1 + size(\varphi) \end{aligned}$$

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Definition 4.6 (*Measure on satisfaction judgements*)
Recalling the definition of $[\![\mathscr{D}]\!]^{(n)}$ from Lemma 3.1, we write $\vec{\mathsf{J}}$ for a finite list of satisfaction judgements of the form $([\![\mathscr{D}]\!]^{(n_i)}, V) \models \varphi_i$, where the inductive definition $\mathscr{D}$ and the valuation V are the same in each judgement. For each natural number n we define $size_{\vec{\mathsf{J}}}(n)$ as follows:

$$size_{\vec{\mathsf{J}}}(n) \quad \equiv \quad \sum_{(([\![\mathscr{D}]\!]^{(n)}, V) \models \varphi) \in \vec{\mathsf{J}}} size(\varphi)$$

Now we write $\mu(\vec{\mathsf{J}})$ for the *multiset* of natural numbers which includes n with multiplicity $size_{\vec{\mathsf{J}}}(n)$, for every $n \in \mathbb{N}$.

Intuitively, $size_{\vec{\mathsf{J}}}(n)$ records the total size of all formulae for which a satisfaction judgement using $[\![\mathscr{D}]\!]^{(n)}$ exists in $\vec{\mathsf{J}}$. The measure $\mu(\vec{\mathsf{J}})$ records a mapping from numbers n to these total sizes. Note that since $\vec{\mathsf{J}}$ is finite, it follows that there are only finitely many n with non-zero multiplicity in $\mu(\vec{\mathsf{J}})$. Hence, we can use the multiset ordering construction of Dershowitz & Manna (1979) to derive a well-founded ordering $\prec$ on $\mu(\vec{\mathsf{J}})$ in terms of the $<$ ordering on natural numbers. We now proceed to the main intermediate result in our proof of logical completeness, where we show that the set of formula reduction steps possible from a given configuration accounts for all satisfying valuations of that configuration. Moreover, we show that our well-founded measure on the list of satisfaction judgements strictly decreases across the formula reduction step.

Lemma 4.5 (*Intermediate lemma for logical completeness*)
Let $\mathscr{D} = \mathtt{fun}\, r(x : S_r) : \mathtt{prop} = \psi$ be an α-inductive definition as in Definition 4.4, and suppose that $\Delta \vdash \overline{c}$ & $\vec{\varphi}$ & $\varphi : \mathtt{prop}$ and $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ both hold. Suppose that $\vec{\varphi} = \varphi_1$ & $\cdots$ & φ_k and that $V \models \overline{c}$, $\forall i \in \{1,\ldots,k\}$. $([\![\mathscr{D}]\!]^{(n_i)}, V) \models \varphi_i$ and $([\![\mathscr{D}]\!]^{(n)}, V) \models \varphi$ all hold, for some $n_1,\ldots,n_k,n$. Then, either

1. $\vec{\varphi} = \emptyset$ and $\varphi = \mathsf{T}$; or
2. there exist Δ', $\overline{c}'$, $\vec{\varphi}'$, φ', V', $n'_1,\ldots,n'_j$ and n' such that

$$\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \exists\Delta,\Delta'(\overline{c}'; \vec{\varphi}'; \varphi') \tag{5}$$

$$V' \models \overline{c}' \tag{6}$$

$$\forall i \in \{1,\ldots,j\}.\ ([\![\mathscr{D}]\!]^{(n'_i)}, V) \models \varphi'_i \tag{7}$$

$$([\![\mathscr{D}]\!]^{(n')}, V') \models \varphi' \tag{8}$$

all hold, where $\vec{\varphi}' = \varphi'_1$ & $\cdots$ & φ'_j and V' is an extension of V to $dom(\Delta,\Delta')$. Furthermore, if $\vec{\mathsf{J}}$ denotes $([\![\mathscr{D}]\!]^{(n_1)}, V) \models \varphi_1, \ldots, ([\![\mathscr{D}]\!]^{(n_k)}, V) \models \varphi_k, ([\![\mathscr{D}]\!]^{(n)}, V) \models \varphi$ and $\vec{\mathsf{J}}'$ stands for $([\![\mathscr{D}]\!]^{(n'_1)}, V') \models \varphi'_1, \ldots, ([\![\mathscr{D}]\!]^{(n'_j)}, V') \models \varphi'_j, ([\![\mathscr{D}]\!]^{(n')}, V') \models \varphi'$ then $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds also.

Proof
The proof is by case analysis on φ – the cases are as follows. We note that the case where $\varphi = \mathsf{F}$ cannot arise since $(\mathfrak{R}, V) \models \mathsf{F}$ is not derivable for any $\mathfrak{R}$, V.

Case $\varphi = r\,p$. In this case we use formula reduction rule (F1) to deduce that Equation (5) holds, where $\Delta' = \emptyset$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \psi[p/x]$. If we set $V' = V$ and $n'_1 = n_1,\ldots,n'_j = n_j$, it is easy to see that both Equations (6) and (7) hold.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Since $(\llbracket\mathscr{D}\rrbracket^{(n)}, V) \models r\,p$, it follows that $n = n' + 1$ for some n' (since $\llbracket\mathscr{D}\rrbracket^{(0)} = \emptyset$ and $(\emptyset, V) \models r\,p$ is not derivable). Then from $(\llbracket\mathscr{D}\rrbracket^{(n'+1)}, V) \models r\,p$ we get $\llbracket p\rrbracket_V \in \llbracket\mathscr{D}\rrbracket^{(n'+1)}$. Using the fixed-point definition of denotations from Definition 3.10 and the definition of $\llbracket\mathscr{D}\rrbracket^{(n'+1)}$, we get that $(\llbracket\mathscr{D}\rrbracket^{(n')}, \{x \mapsto \llbracket p\rrbracket_V\}) \models \psi$. From Lemma 4.1(3) we get $(\llbracket\mathscr{D}\rrbracket^{(n')}, V[x \mapsto \llbracket p\rrbracket_V]) \models \psi$, and by Lemma 4.2 it follows that $(\llbracket\mathscr{D}\rrbracket^{(n')}, V) \models \psi[p/x]$, i.e. Equation (8) holds.

Finally, note that the multiset $\mu(\vec{\mathsf{J}}')$ is obtained from the multiset $\mu(\vec{\mathsf{J}})$ by replacing $size(r\,p) = 1$ occurrence of $n = n' + 1$ with $size(\psi[p/x])$ occurrences of n'; hence $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$.

Case $\varphi = c$. In this case our assumption tells us that $V \models \overline{c}\ \&\ c$, i.e. $\models \exists\Delta(\overline{c}\ \&\ c)$. Then by (F2) we get that Equation (5) holds, where $\Delta' = \emptyset$, $\overline{c}' = \overline{c}\ \&\ c$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \mathsf{T}$. If we set $V' = V$, $n' = n$, $n'_1 = n_1, \ldots, n'_j = n_j$ it follows that Equations (6) and (7) both hold. We get that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds because the multiplicity of n decreases by one, since $size(\mathsf{T}) < size(c)$.

Case $\varphi = \mathsf{T}$. In this case we perform a case split on the goal list $\vec{\varphi}$. If $\vec{\varphi} = \emptyset$ then we are immediately done, so we consider the case where $\vec{\varphi} = \vec{\varphi}'', \varphi''$ for some $\vec{\varphi}''$ and φ'', i.e. $k = k' + 1$ for some k'. In this case, by (F3) we get that Equation (5) holds, where $\Delta' = \emptyset$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}''$ (i.e. $j = k'$) and $\varphi' = \varphi''$. Then if we set $V' = V$, $n' = n_k$ and $n'_1 = n_1, \ldots, n'_j = n_j$ we get that Equations (6) and (7) both hold, by assumption. Finally, since the T formula has been eliminated completely, it follows that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, as required.

Case $\varphi = \varphi_1\ \&\ \varphi_2$. In this case we get that $(\llbracket\mathscr{D}\rrbracket^{(n)}, V) \models \varphi_m$ holds, for all $m \in \{1,2\}$. Then by rule (F4) we get that Equation (5) holds, where $\Delta' = \emptyset$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}, \varphi_2$ (i.e. $j = k + 1$) and $\varphi' = \varphi_1$. Then Equations (6) and (7) both hold if we set $V' = V$, $n' = n$, $n'_1 = n_1, \ldots, n'_k = n_k$ and $n'_j = n$. Since $size(\varphi_1) + size(\varphi_2) < size(\varphi_1\ \&\ \varphi_2)$, it follows that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, as required.

Case $\varphi = \varphi_1 \vee \varphi_2$. Here we get that $(\llbracket\mathscr{D}\rrbracket^{(n)}, V) \models \varphi_m$ holds, for some $m \in \{1,2\}$. Then by rule (F5) we get that Equation (5) holds when $\Delta' = \emptyset$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \varphi_m$. If we set $V' = V$, $n' = n$ and $n'_1 = n_1, \ldots, n'_k = n_k$ then both Equations (6) and (7) hold. Finally, since $size(\varphi_j) < size(\varphi_1 \vee \varphi_2)$, we get that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, as required.

Case $\varphi = \exists x : E.\, \varphi''$. By α-renaming the bound variable in the formula we can assume that $x \notin dom(\Delta)$. Then we get that $(\llbracket\mathscr{D}\rrbracket^{(n)}, V[x \mapsto t]) \models \varphi''$, for some $t \in \alpha\text{-}Tree_{\Sigma}(E)$. By rule (F6) we get that Equation (5) holds if we let $\Delta = \{x : E\}$, $\overline{c}' = \overline{c}$, $\vec{\varphi}' = \vec{\varphi}$ (i.e. $j = k$) and $\varphi' = \varphi''$. If we also let $V' = V[x \mapsto t]$, $n' = n$ and $n'_1 = n_1, \ldots, n'_k = n_k$, we can use Lemma 4.1(3) to show that Equations (6) and (7) both hold. Also, it follows that $\mu(\vec{\mathsf{J}}') \prec \mu(\vec{\mathsf{J}})$ holds, since $size(\varphi'') < size(\exists x : E.\, \varphi'')$.

This completes the proof of Lemma 4.5. □

Our proof of logical completeness rests on the fact that if $(\llbracket\mathscr{D}\rrbracket, V) \models \varphi$ holds then we can unfold the inductive definition, $\mathscr{D}$, n times (for some n) to produce an α-tree relation $\llbracket\mathscr{D}\rrbracket^{(n)}$ such that $(\llbracket\mathscr{D}\rrbracket^{(n)}, V) \models \varphi$ holds.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Lemma 4.6 (*Unfolding α-inductive definitions*)
Let $\mathscr{D} = \mathtt{fun}\, r(x : S_r) : \mathtt{prop} = \psi$ be an α-inductive definition as in Definition 4.4, and suppose that $\Delta \vdash \varphi : \mathtt{prop}$ and $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ both hold. Then $([\![\mathscr{D}]\!], V) \models \varphi$ holds iff $([\![\mathscr{D}]\!]^{(n)}, V) \models \varphi$ holds for some n.

Proof
We proceed by induction on the structure of φ, using the fact that $[\![\mathscr{D}]\!] = \bigcup_{n\in\mathbb{N}} [\![\mathscr{D}]\!]^{(n)}$ (by Lemma 3.1). In the case of an atomic formula $r\,p$, we observe that if $(\mathscr{D}, V) \models r\,p$ then there exists n such that $\psi[p/x] \in [\![\mathscr{D}]\!]^{(n)}$, i.e. $([\![\mathscr{D}]\!]^{(n)}, V) \models r\,p$. In the case of a conjunction φ_1 & φ_2 we use the fact that $[\![\mathscr{D}]\!]^{(n)} \subseteq [\![\mathscr{D}]\!]^{(n+1)}$ (which follows from the definition of $\Phi_{\mathscr{D}}$) to obtain a value n which is high enough to satisfy both φ_1 and φ_2. The other cases are straightforward. □

We now use Lemma 4.5 and Definition 4.4, along with Lemma 4.6, to present a proof of logical completeness.

Lemma 4.7 (*Logical completeness*)
With $\mathscr{D}$ and $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ as in Definition 4.4, for all $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ it is the case that if $([\![\mathscr{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$ then there exists some $\exists\Delta'(\overline{c}') \in solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $([\![\mathscr{D}]\!], V) \models \exists\Delta'(\overline{c}')$.

Proof
With the same assumptions as Definition 4.4, given an α-tree valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ we assume that $([\![\mathscr{D}]\!], V) \models \overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi$. It follows from Lemma 4.6 that we can find a list of satisfaction judgements $\vec{\mathtt{J}}$ as in the hypothesis of Lemma 4.5. Then by Lemma 4.5 we can build up a sequence of formula reductions $\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots$ with associated lists of satisfaction judgements $\vec{\mathtt{J}}$ of the form described in Definition 4.6. From Lemma 4.5 we know that at each step $\mu(\vec{\mathtt{J}})$ decreases in the well-founded ordering defined above, therefore the sequence cannot be infinite and must eventually terminate. Furthermore, by Lemma 4.5 that finite reduction sequence must be of the form $\mathscr{D} \vdash \exists\Delta(\overline{c}; \vec{\varphi}; \varphi) \rightsquigarrow \cdots \rightsquigarrow \exists\Delta, \Delta'(\overline{c}'; \emptyset; \mathsf{T})$, where $V' \models \overline{c}'$ for some V', which extends V to $dom(\Delta, \Delta')$. Thus, by Definition 4.4 we have shown that there is some $\exists\Delta'(\overline{c}') \in solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ such that $([\![\mathscr{D}]\!], V) \models \exists\Delta'(\overline{c}')$ holds, as required. □

4.4 Correctness of formula reduction

We now state the theorem which relates the semantics of schematic formulae and the operational semantics of αML, which follows immediately from Lemmas 4.4 and 4.7.

Theorem 4.7 (*Correctness of formula reduction*)
With $\mathscr{D}$ and $(\Delta, \overline{c}, \vec{\varphi}, \varphi)$ as in Definition 4.4, for all $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ it is the case that $(\exists\Delta'(\overline{c}') \in solns_{\mathscr{D}}(\Delta, \overline{c}, \vec{\varphi}, \varphi) \wedge ([\![\mathscr{D}]\!], V) \models \exists\Delta'(\overline{c}')) \iff ([\![\mathscr{D}]\!], V) \models (\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$ holds, for some Δ' and $\overline{c}'$.

Thus, the operational semantics of αML computes all and only the solutions to an initial query formula $(\overline{c}\ \&\ \vec{\varphi}\ \&\ \varphi)$, expressed as an αML formula reduction configuration $\exists\Delta(\overline{c}; \vec{\varphi}; \varphi)$. The relative simplicity of these proofs is a demonstration of the power of Theorem 4.3 because when doing proofs about embedded formulae we can forget about the details of the αML operational semantics which are not relevant and just focus on the subset of formula reduction transitions.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

5 Contextual equivalence of formulae and α-inductive definitions

The *logical soundness and completeness* results from Theorem 4.7 give us the following weak result about equivalence of encoded formulae in αML.

Corollary 5.1 (*Equivalence for CLP goal states*)
If $\mathscr{D} \models \forall \Delta.\, \varphi \equiv \varphi'$ then $\exists \Delta'(\overline{c}; F_{\vec{\varphi}}; \varphi[v_{\mathscr{D}}/r])\downarrow \iff \exists \Delta'(\overline{c}; F_{\vec{\varphi}}; \varphi'[v_{\mathscr{D}}/r])\downarrow$ holds for any $\Delta' \supseteq \Delta$, any $\overline{c}$ and any frame stack $F_{\vec{\varphi}}$ which corresponds to a CLP goal list in the sense of Definition 4.2.

This holds because if the formulae φ and φ' have the same semantics then $\overline{c}$ & $\vec{\varphi}$ & φ and $\overline{c}$ & $\vec{\varphi}$ & φ' also have the same semantics. Then Corollary 5.1 follows straightforwardly from Theorem 4.7.

In the remainder of this section we prove more general results than Corollary 5.1, concerning the behaviour of encoded schematic formulae and α-inductive definitions (such as the definition from Example 4.1) in *arbitrary* αML contexts. We will assume that both $\emptyset \vdash v_{\mathscr{D}} : S_r \to \texttt{prop}$ and $\emptyset \vdash v_{\mathscr{D}'} : S_r \to \texttt{prop}$ hold, where $v_{\mathscr{D}}$ and $v_{\mathscr{D}'}$ are the αML encodings of the α-inductive definitions $\mathscr{D}$ and $\mathscr{D}'$, respectively.

5.1 Contextually equivalent formulae have the same semantics

We begin by showing that contextual equivalence implies semantic equivalence for encoded formulae. This is one area of the theory where the proofs rely on our underlying nominal sets model of abstract syntax with binders. Firstly, however, we define a class of αML expressions which encode α-tree valuations in the following sense.

Definition 5.2 (*Characteristic expressions*)
Suppose that the α-tree valuation $V = \{x_1 \mapsto [g_1]_\alpha, \ldots, x_n \mapsto [g_n]_\alpha\}$. Then we write e_V to stand for a *characteristic expression* of V, which is any expression of the form

$$\texttt{let}\, z_1 = [\![g_1]\!]\, \texttt{in} \cdots \texttt{in}\, \texttt{let}\, z_n = [\![g_n]\!]\, \texttt{in}\, x_1 = z_1\ \&\ \cdots\ \&\ x_n = z_n$$

where the bound variables $z_1, \ldots, z_n$ are pairwise distinct, are disjoint from $x_1, \ldots, x_n$ and do not occur free in any of the tree translations $[\![g_i]\!]$. These restrictions can always be satisfied by α-renaming the `let`-bound variables.

Note that it does not matter which representatives of the α-equivalence class we choose, because Property 2.12 guarantees that α-equivalent ground trees will be translated into contexually equivalent αML expressions.

Lemma 5.1 (*Typing for characteristic expressions*)
Suppose that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ has the form of Definition 5.2 above, and that e_V is a characteristic expression of V. Then $\Delta' \vdash e_V : \texttt{prop}$ holds for any $\Delta' \supseteq \Delta$ such that $\Delta'(\mathscr{V}(n)) = sort(n)$ for all $n \in FN(g_1, \ldots, g_n)$.

We now consider the behaviour of characteristic expressions when they are evaluated. Recall that the definition of a characteristic expression e_V of a valuation $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ involves translated ground trees $[\![g_i]\!]$. A free name n of the tree g_i is translated using the fixed bijection $\mathscr{V}$, and corresponds to a free variable $\mathscr{V}(n)$ of $[\![g_i]\!]$, and hence of e_V. Since the bijection $\mathscr{V}(-)$ that maps names to variables is fixed, it is possible that one of the

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

free variables $\mathscr{V}(n)$ could clash with a variable from the typing environment $dom(\Delta)$, which represents the types of variables generated by the rest of the computation. This is problematic because the variables which are used to represent the ground trees $[\![g_i]\!]$ are not related to the variables which appear in $dom(\Delta)$. In the following lemmas we will assume that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ of the form $\{x_1 \mapsto [g_1]_\alpha, \ldots, x_n \mapsto [g_n]_\alpha\}$ is such that following property holds:

$$\{\mathscr{V}(n) \mid n \in FN(g_1, \ldots, g_n)\} \cap \{x_1, \ldots, x_n\} = \emptyset \tag{9}$$

In the proof of the main theorem in this section (Theorem 5.3) we will use an argument based on equivariance to show that this problem can be avoided.

Lemma 5.2 (*Evaluation of characteristic expressions*)
Suppose that $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ as in Definition 5.2 and has the property (9). Pick arbitrary Δ', F and T such that $\Delta' \supseteq \Delta$, $\Delta' \vdash FN(g_1, \ldots, g_n)$ and $\Delta' \vdash F : \mathtt{prop} \to T$ all hold. Then there exist η_V and $\overline{c}_V$ such that

$$\exists \Delta'(\mathsf{T}; F; e_V) \longrightarrow \cdots \longrightarrow \exists \Delta', \eta_V(\overline{c}_V; F; \mathsf{T}) \tag{10}$$

and $\models \exists \Delta', \eta_V(\overline{c}_V)$ both hold. Furthermore, for any $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$, if $V^* \models \overline{c}_V$ then there exists a permutation π^* such that $V^*(x) = \pi^* \cdot V(x)$ for all $x \in dom(\Delta)$.

Proof
We know the form of the expression e_V from Definition 5.2. Since the evaluation of the ground trees $[\![g_i]\!]$ only produces freshness constraints, it follows that evaluating the ground trees will succeed: now suppose that evaluating $[\![g_i]\!]$ produces η_i, $\overline{c}_i$ and v_i. Therefore, we get that

$$\exists \Delta'(\mathsf{T}; F; e_V) \longrightarrow \cdots \longrightarrow \exists \Delta', \eta_1, \ldots, \eta_n(\overline{c}_1 \,\&\, \cdots \,\&\, \overline{c}_n; F; x_1 = v_1 \,\&\, \cdots \,\&\, x_n = v_n) \tag{11}$$

Since we have assumed that V has the property (9), it follows that the assignments to the variables $x_1, \ldots, x_n$ in the second configuration of Equation (11) cannot conflict with the freshness constraints $\overline{c}_1 \,\&\, \cdots \,\&\, \overline{c}_n$. Therefore, from Equation (11) it follows that Equation (10) holds, where $\eta_V = \eta_1, \ldots, \eta_n$ and $\overline{c}_V = \overline{c}_1 \,\&\, \cdots \,\&\, \overline{c}_n \,\&\, x_1 = v_1 \,\&\, \cdots \,\&\, x_n = v_n$. Because the constraints in $\overline{c}_1, \ldots, \overline{c}_n$ are all freshnesses and the variables $x_1, \ldots, x_n$ do not appear elsewhere, it follows that $\models \exists \Delta', \eta_V(\overline{c}_V)$.

Now suppose that $V^* \in \alpha\text{-}Tree_\Sigma(\Delta', \eta_V)$ is such that $V^* \models \overline{c}_V$. By the semantics of constraints it follows that $V^*(x) = [\![v_i]\!]_{V^*}$ holds for all $i \in \{1, \ldots, n\}$. Now there exists a permutation π^* such that $(\pi^* \cdot g_i) \in [\![v_i]\!]_{V^*}$ for all $i \in \{1, \ldots, n\}$, and finally, since $V(x_i) = [g_i]_\alpha$ by its definition from Definition 5.2, it follows that $V^*(x_i) = \pi^* \cdot V(x_i)$ for all $i \in \{1, \ldots, n\}$, as required. □

The previous lemma formalised the sense in which a characteristic expression e_V represents the α-tree valuation V. We now prove the central lemma of this proof, in which we show that the expression $e_V \,\mathtt{then}\, \varphi[v_{\mathscr{D}}/r]$ terminates in the empty context iff $([\![\mathscr{D}]\!], V) \models \varphi$. Here we write $e_1 \,\mathtt{then}\, e_2$ as syntactic sugar for the sequential evaluation of e_1 followed by e_2, which can be implemented in αML as $\mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2$, where $x \notin FV(e_2)$.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Lemma 5.3 (*Intermediate result concerning termination and satisfaction*)
Suppose that $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$ has the form of Definition 5.2 above, and satisfies the property (9). We pick an arbitrary Δ' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash FN(g_1, \ldots, g_n)$ both hold. Let $\mathscr{D}$ be an arbitrary α-inductive definition (in the standard form) and let φ be a schematic formula such that $\Delta' \vdash \varphi[v_{\mathscr{D}}/r] : \mathtt{prop}$ holds. Then it is the case that $\exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r])\downarrow$ iff $([\![\mathscr{D}]\!], V) \models \varphi$.

Proof
By Lemma 5.2, $\exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r]) \longrightarrow \cdots \longrightarrow \exists\Delta', \eta_V(\overline{c}_V; \mathtt{Id}; \varphi[v_{\mathscr{D}}/r])$ and $\models \exists\Delta', \eta_V(\overline{c}_V)$ both hold. Furthermore, for any $V^* \in \alpha\text{-}Tree_{\Sigma}(\Delta', \eta_V)$, if $V^* \models \overline{c}_V$ then there exists a permutation π^* such that $V^*(x) = \pi^* \cdot V(x)$ for all $x \in dom(\Delta)$. Now we prove the two directions separately.

$\exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r])\downarrow \Longrightarrow ([\![\mathscr{D}]\!], V) \models \varphi$**.**
From $\exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r])$ we get that $\exists\Delta', \eta_V(\overline{c}_V; \mathtt{Id}; \varphi[v_{\mathscr{D}}/r])\downarrow$ holds, from which it follows that there exist Δ_φ and $\overline{c}_\varphi$ such that $\exists\Delta_\varphi(\overline{c}_\varphi) \in solns_{\mathscr{D}}((\Delta', \eta_V), \overline{c}_V, \emptyset, \varphi)$ and $V^* \models \exists\Delta_\varphi(\overline{c}_\varphi)$ both hold, for some $V^* \in \alpha\text{-}Tree_{\Sigma}(\Delta', \eta_V)$. It follows that $V^* \models \overline{c}_V$, and hence that $V^*(x) = \pi^* \cdot V(x)$ holds, for all $x \in dom(\Delta)$ and for some permutation π^*. By Logical Soundness (Theorem 4.7) we get that $([\![\mathscr{D}]\!], V^*) \models \overline{c}_V \ \&\ \varphi$ holds, which we simplify to $([\![\mathscr{D}]\!], V^*) \models \varphi$ or, equivalently, $([\![\mathscr{D}]\!], \pi^* \cdot V) \models \varphi$. Finally, from equivariance it follows that $([\![\mathscr{D}]\!], V) \models \varphi$ holds, as required.

$([\![\mathscr{D}]\!], V) \models \varphi \Longrightarrow \exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r])$**.**
We can deduce that there exists a permutation π^* and a valuation $V^* \in \alpha\text{-}Tree_{\Sigma}(\Delta', \eta_V)$ such that $V^* \models \overline{c}_V$ and $V^*(x) = \pi^* \cdot V(x)$ hold for all $x \in dom(\Delta)$. Now, since formula satisfaction and denotations of α-inductive definitions are both equivariant, from our initial assumption that $([\![\mathscr{D}]\!], V) \models \varphi$ we conclude that $([\![\mathscr{D}]\!], \pi^* \cdot V) \models \varphi$. Thus, we get that $V^* \models \overline{c}_V \ \&\ \varphi$ holds. Using this fact, along with the Logical Completeness result from Theorem 4.7, we get that there exist Δ_φ and $\overline{c}_\varphi$ such that $\exists\Delta_\varphi(\overline{c}_\varphi) \in solns_{\mathscr{D}}((\Delta', \eta_V), \overline{c}_V, \emptyset, \overline{c})$ and $V^* \models \exists\Delta_\varphi(\overline{c}_\varphi)$. Hence, $\exists\Delta'(\mathsf{T}; \mathtt{Id}; e_V \mathtt{\ then\ } \varphi[v_{\mathscr{D}}/r])\downarrow$ holds, as required.

This completes the proof of Lemma 5.3. □

The proof of Lemma 5.3 relies on the fact that ground trees can be represented in αML in a way that respects α-equivalence. Specifically, it relies on details of the underlying semantics of schematic formulae in terms of α-equivalence classes of ground trees (which form a nominal set, as shown in Section 3.4). Arguments based on equivariance are typical in the world of nominal sets and nominal logic (Pitts, 2003, 2006) but are largely absent from this paper. We consider it advantageous that the details of the underlying mathematical model are hidden from view to such a degree.

We now prove the main result of this section that if two formulae are contextually equivalent then they have the same semantics, using equivariance to argue that it is sufficient to consider valuations which have property (9).

Theorem 5.3 (*Contextual equivalence implies semantic equivalence*)
For all $\mathscr{D}$, $\mathscr{D}'$, Δ, φ and φ', if $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$ then $\mathscr{D}, \mathscr{D}' \models \forall\Delta.\, \varphi \equiv \varphi'$.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Proof
We assume that $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$ and pick any $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$: we must show that $([\![\mathscr{D}]\!], V) \models \varphi \iff ([\![\mathscr{D}']\!], V) \models \varphi'$. By the equivariance property this is equivalent to $([\![\mathscr{D}]\!], \pi \cdot V) \models \varphi \iff ([\![\mathscr{D}']\!], \pi \cdot V) \models \varphi'$ for any permutation π. Now if V does not have the disjointness property (9) then we can always find a suitable permutation π to produce a valuation $\pi \cdot V$, which does have that property. Therefore, it is sufficient to consider valuations which satisfy property (9). Now we pick an arbitrary type environment Δ' such that $\Delta' \supseteq \Delta$ and $\Delta' \vdash FN(g_1, \ldots, g_n)$ both hold. Then, since $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$ we know in particular that

$$\exists \Delta, \eta(\mathsf{T}; \mathtt{Id}; e_V \ \mathtt{then}\ \varphi[v_{\mathscr{D}}/r])\downarrow \iff \exists \Delta, \eta(\mathsf{T}; \mathtt{Id}; e_V \ \mathtt{then}\ \varphi'[v_{\mathscr{D}'}/r])\downarrow.$$

Then by Lemma 5.3 it follows that $([\![\mathscr{D}]\!], V) \models \varphi \iff ([\![\mathscr{D}']\!], V) \models \varphi'$. Thus, we have shown that $\mathscr{D}, \mathscr{D}' \models \forall \Delta.\, \varphi \equiv \varphi'$ holds, as required. □

5.2 Formulae with the same semantics are contextually equivalent

We now show that if two formulae have the same semantics then they have identical termination behaviour in any αML evaluation context. This direction of the proof is somewhat delicate, and relies on certain properties of the αML reduction relation. We first prove a preparatory lemma, which states that two configurations which differ only in their constraints, but share satisfying valuations for their shared existential variables, have similar termination behaviour. This is a general property of all expressions, not just of the expressions corresponding to formulae which concern us in this section.

Below we write $V \models \exists \Delta'(\overline{c})$ to mean that $V \in \alpha\text{-}Tree_{\Sigma}(\Delta)$, $dom(\Delta) \cap dom(\Delta') = \emptyset$ and $\Delta \vdash \exists \Delta'(\overline{c}) : \mathtt{prop}$ all hold, and that there exists $V' \in \alpha\text{-}Tree_{\Sigma}(\Delta')$ such that $(V, V') \models \overline{c}$. Therefore, V comprises part of a satisfying instantiation for $\overline{c}$.

Lemma 5.4 (*Preparatory lemma*)
Suppose that Δ', Δ_1 and Δ_2 have pairwise disjoint domains, i.e. $dom(\Delta')$ contains precisely those variables shared between Δ', Δ_1 and Δ', Δ_2. Assume that $\emptyset \vdash \exists \Delta', \Delta_1(\overline{c}_1; F; e) : T$ and $\emptyset \vdash \exists \Delta', \Delta_2(\overline{c}_2; F; e) : T$ both hold, and that, for all $V' \in \alpha\text{-}Tree_{\Sigma}(\Delta')$, if $V' \models \exists \Delta_1(\overline{c}_1)$ then $V' \models \exists \Delta_2(\overline{c}_2)$. Then $\exists \Delta', \Delta_1(\overline{c}_1; F; e)\downarrow$ implies $\exists \Delta', \Delta_2(\overline{c}_2; F; e)\downarrow$.

Proof
We assume that $\exists \Delta', \Delta_1(\overline{c}_1; F; e)\downarrow$ holds and proceed by well-founded induction on the number of steps in this derivation. The assumption that $V' \models \exists \Delta_1(\overline{c}_1)$ implies $V' \models \exists \Delta_2(\overline{c}_2)$ for any $V' \in \alpha\text{-}Tree_{\Sigma}(\Delta')$ is necessary for those cases where the constraint set is augmented. In cases where additional existential variables are generated, by α-renaming we may assume that these are distinct from all the variables in $dom(\Delta')$, $dom(\Delta_1)$ *and* $dom(\Delta_3)$. □

We can now present a proof of the main result in this section, which is that semantically equivalent formulae are always contextually equivalent in the meta-language.

Theorem 5.4 (*Formulae with same semantics are contextually equivalent*)
For all $\mathscr{D}$, $\mathscr{D}'$, Δ, φ, φ', if $\mathscr{D}, \mathscr{D}' \models \forall \Delta.\, \varphi \equiv \varphi'$ then $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

Proof
We assume that $\mathscr{D}, \mathscr{D}' \models \forall \Delta.\, \varphi \equiv \varphi'$, and pick arbitrary Δ', $\overline{c}$, F and T such that $\Delta' \supseteq \Delta$, $\Delta' \vdash \overline{c} : \mathtt{prop}$ and $\Delta' \vdash F : \mathtt{prop} \to T$. We deduce that $\mathscr{D}, \mathscr{D}' \models \forall \Delta'.\, \overline{c} \,\&\, \varphi \equiv \overline{c} \,\&\, \varphi'$ also holds. Now, we assume that $\exists \Delta'(\overline{c}; F; \varphi[v_{\mathscr{D}}/r])\downarrow$. It follows that

$$\exists \Delta'(\overline{c}; F; \varphi[v_{\mathscr{D}}/r]) \longrightarrow \cdots \longrightarrow \exists \Delta', \Delta_1(\overline{c} \,\&\, \overline{c}_1; F; \mathsf{T})$$

and also that $\models \exists \Delta', \Delta_1(\overline{c} \,\&\, \overline{c}_1)$ and $\exists \Delta', \Delta_1(\overline{c} \,\&\, \overline{c}_1; F; \mathsf{T})\downarrow$, for some $\Delta_1, \Delta_2, \overline{c}_1, \overline{c}_2$. We get that $\exists \Delta'(\overline{c}; \mathtt{Id}; \varphi[v_{\mathscr{D}}/r]) \longrightarrow \cdots \longrightarrow \exists \Delta', \Delta_1(\overline{c} \,\&\, \overline{c}_1; \mathtt{Id}; \mathsf{T})$, i.e. $\exists \Delta'(\overline{c}; \mathtt{Id}; \varphi[v_{\mathscr{D}}/r])\downarrow$. Thus, we get that $\exists \Delta'(\overline{c} \,\&\, \overline{c}_1) \in \mathit{solns}_{\mathscr{D}}(\Delta_1, \overline{c}, \emptyset, \varphi)$.

Since $\models \exists \Delta', \Delta_1(\overline{c} \,\&\, \overline{c}_1)$, there exist $V' \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta')$ and $V_1 \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta_1)$ such that $(V', V_1) \models \overline{c} \,\&\, \overline{c}_1$ holds, i.e. $V' \models \exists \Delta_1(\overline{c} \,\&\, \overline{c}_1)$. By Logical Soundness (Theorem 4.7) it follows that $([\![\mathscr{D}]\!], V') \models \overline{c} \,\&\, \varphi$. Thus, we get $([\![\mathscr{D}']\!], V') \models \overline{c} \,\&\, \varphi'$, and by Logical Completeness (Theorem 4.7) there exists $\exists \Delta_2(\overline{c}'_2) \in \mathit{solns}_{\mathscr{D}'}(\Delta', \overline{c}, \emptyset, \varphi')$ such that $V' \models \exists \Delta_2(\overline{c}'_2)$. Since the variables in Δ_2 are introduced by existential quantification, by α-renaming we may assume, without loss of generality, that $\mathit{dom}(\Delta_2)$ is disjoint from $\mathit{dom}(\Delta)$ and $\mathit{dom}(\Delta_1)$. We know that $\overline{c}'_2 \equiv \overline{c} \,\&\, \overline{c}_2$ for some $\overline{c}_2$, and that

$$\exists \Delta'(\overline{c}; \mathtt{Id}; \varphi'[v_{\mathscr{D}'}/r]) \longrightarrow \cdots \longrightarrow \exists \Delta', \Delta_2(\overline{c} \,\&\, \overline{c}_2; \mathtt{Id}; \mathsf{T})$$

and $\models \exists \Delta', \Delta_2(\overline{c} \,\&\, \overline{c}_2)$ both hold. It follows that

$$\exists \Delta'(\overline{c}; F; \varphi'[v_{\mathscr{D}'}/r]) \longrightarrow \cdots \longrightarrow \exists \Delta', \Delta_2(\overline{c} \,\&\, \overline{c}_2; F; \mathsf{T}).$$

Note that Δ_1 and Δ_2 can be assumed to have disjoint domains, and that these can be assumed to be disjoint from $\mathit{dom}(\Delta')$. Furthermore, the above argument shows that $V^* \models \exists \Delta_1(\overline{c} \,\&\, \overline{c}_1)$ implies $V^* \models \exists \Delta_2(\overline{c} \,\&\, \overline{c}_2)$ for an arbitrary $V^* \in \alpha\text{-}\mathit{Tree}_\Sigma(\Delta')$. Therefore, by Lemma 5.4 we get that $\exists \Delta', \Delta_2(\overline{c} \,\&\, \overline{c}_2; F; \mathsf{T})\downarrow$, and hence that $\exists \Delta'(\overline{c}; F; \varphi'[v_{\mathscr{D}'}/r])\downarrow$.

By a similar argument we get that $\exists \Delta'(\overline{c}; F; \varphi'[v_{\mathscr{D}'}/r])\downarrow$ implies $\exists \Delta'(\overline{c}; F; \varphi[v_{\mathscr{D}}/r])\downarrow$. Therefore, we have $\exists \Delta'(\overline{c}; F; \varphi[v_{\mathscr{D}}/r])\downarrow \iff \exists \Delta'(\overline{c}; F; \varphi'[v_{\mathscr{D}'}/r])\downarrow$, and hence it follows that $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$ holds, as required. □

5.3 Contextual equivalence and semantic equivalence for schematic formulae

We can now state a key result about contextual equivalence of encoded schematic formulae in arbitrary αML contexts, which follows from Theorems 5.3 and 5.4.

Theorem 5.5 (*Contextual equivalence for schematic formulae*)
For all $\mathscr{D}$, $\mathscr{D}'$, Δ, φ and φ' it is the case that $\Delta \vdash \varphi[v_{\mathscr{D}}/r] \cong \varphi'[v_{\mathscr{D}'}/r] : \mathtt{prop}$ holds iff $\mathscr{D}, \mathscr{D}' \models \forall \Delta.\, \varphi \equiv \varphi'$ holds.

This result is interesting because the full range of contexts in αML are richer than just CLP goal states – in particular, the presence of higher order functions means that formulae may be packaged up inside a function and passed around before eventually being evaluated. In this setting it is by no means obvious that semantically equivalent formulae always have the same behaviour with regard to termination. Furthermore, this result relates the behaviour of formulae which are equivalent but with regard to different α-inductive definitions.

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

5.4 Contextual equivalence of encoded α-inductive definitions

In this section we deduce a final result, which concerns contextual equivalence of the recursive function values which denote α-inductive definitions. Firstly, however, we must prove an extensional equality result concerning recursive function values in αML.

Lemma 5.5 (*Extensional equality for recursive functions*)
For all Δ, v_1, v_2, T and T', $\Delta \vdash v_1 \cong v_2 : T \to T'$ holds iff $\forall v, \Delta'.\ \Delta' \supseteq \Delta \wedge \Delta' \vdash v : T \Longrightarrow \Delta' \vdash v_1\, v \cong v_2\, v : T'$.

Proof
The forward direction follows straightforwardly from the reflexivity and substitutivity properties of the $\cong^\circ$ relation. For the reverse direction, we assume that $\Delta' \vdash v_1\, v \cong v_2\, v : T'$ holds, for any $\Delta' \supseteq \Delta$ and v such that $\Delta' \vdash v : T$ holds. We note that both v_1 and v_2 must be recursive function values of type $T \to T'$, and that it suffices to prove that $\exists\Delta'(\overline{c}; F; v_1)\downarrow$ implies $\exists\Delta'(\overline{c}; F; v_2)\downarrow$. We consider the sequence $\exists\Delta(\overline{c}; F; v_1) \longrightarrow \cdots \longrightarrow \exists\Delta'(\overline{c}'; \mathtt{Id}; v')$, which, by assumption, is finite and has $\models \exists\Delta'(\overline{c}')$.

Since both v_1 and v_2 are recursive function values, by the reduction rules from Figure 3, any reduction sequence starting from $\exists\Delta'(\overline{c}; F; v_1)$ can be mimicked by a similar reduction sequence starting from $\exists\Delta'(\overline{c}; F; v_2)$, *until* a configuration of the form $\exists\Delta^*(\overline{c}^*; F^*; v_1\, v^*)$ is reached. Applying the recursive functions to a value is the only way to distinguish them because the other reduction rules simply pass recursive functions around the program.

Hence, we perform a case split on whether any configuration in that sequence has the form $\exists\Delta^*(\overline{c}^*; F^*; v_1\, v^*)$. If not, then the recursive function value v_1 is never applied and it follows that there exists a corresponding finite reduction sequence $\exists\Delta(\overline{c}; F; v_2) \longrightarrow \cdots \longrightarrow \exists\Delta'(\overline{c}'; \mathtt{Id}; v')$, and since $\models \exists\Delta'(\overline{c}')$, we get that $\exists\Delta'(\overline{c}; F; v_2)\downarrow$ holds, as required. If so, let $\exists\Delta^*(\overline{c}^*; F^*; v_1\, v^*)$ be the first such configuration in the reduction sequence. It follows that $\exists\Delta(\overline{c}; F; v_1) \longrightarrow \cdots \longrightarrow \exists\Delta^*(\overline{c}^*; F^*; v_1\, v^*)$ holds, and since the recursive function v_1 has not been applied before that point, we know that there exists a corresponding finite reduction sequence $\exists\Delta(\overline{c}; F; v_2) \longrightarrow \cdots \longrightarrow \exists\Delta^*(\overline{c}^*; F^*; v_2\, v^*)$, and furthermore $\Delta^* \supseteq \Delta$. By assumption we know that $\Delta^* \vdash v_1\, v^* \cong v_2\, v^* : T'$ holds, and it follows that $\exists\Delta'(\overline{c}; F; v_2)\downarrow$ holds, as required. □

We can now show that two recursive function values (of the same type) which denote α-inductive definitions are contextually equivalent iff the denotations of those definitions are equal, which is a consequence of Theorem 5.5.

Theorem 5.6 (*Contextual equivalence of α-inductive definitions*)
Suppose that $\mathscr{D}$ and $\mathscr{D}'$ are α-inductive definitions in standard form such that $[\![\mathscr{D}]\!] \subseteq S_r$ and $[\![\mathscr{D}']\!] \subseteq S_r$ both hold, for some fixed S_r. Then $[\![\mathscr{D}]\!] = [\![\mathscr{D}']\!]$ iff $\emptyset \vdash v_{\mathscr{D}} \cong v_{\mathscr{D}'} : S_r \to \mathtt{prop}$.

Proof
Let Δ, p and V stand for an arbitrary-type environment, pattern and valuation such that $\Delta \vdash p : S_r$ and $V \in \alpha\text{-}Tree_\Sigma(\Delta)$ both hold. We assume that $[\![\mathscr{D}]\!] = [\![\mathscr{D}']\!]$. By the compactness property of the denotations of α-inductive definitions, this is equivalent to $[\![p]\!]_V \in [\![\mathscr{D}]\!] \Longleftrightarrow [\![p]\!]_V \in [\![\mathscr{D}']\!]$. By the definition of satisfaction, this is equivalent to $([\![\mathscr{D}]\!], V) \models r\, p \Longleftrightarrow ([\![\mathscr{D}']\!], V) \models r\, p$, which is equivalent, by the definition of formula equivalence, to $\mathscr{D}, \mathscr{D}' \models \forall\Delta.\, r\, p \equiv r\, p$. By Theorem 5.5 this is equivalent to $\Delta \vdash (r\, p)[r/v_{\mathscr{D}}] \cong (r\, p)[r/v_{\mathscr{D}'}] : \mathtt{prop}$,

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

and from the definition of capture-avoiding substitution this is equivalent to $\Delta \vdash v_{\mathscr{D}}\, p \cong v_{\mathscr{D}'}\, p : \mathtt{prop}$. Finally, by Lemma 5.5 we may conclude that $[\![\mathscr{D}]\!] = [\![\mathscr{D}']\!]$ is equivalent to $\emptyset \vdash v_{\mathscr{D}} \cong v_{\mathscr{D}'} : S_r \to \mathtt{prop}$, as required. □

6 Related work

6.1 Nominal meta-programming languages

The αML programming language, and the approach to meta-programming described in this paper, descend directly from existing nominal meta-programming languages, in particular FreshML and αProlog.

FreshML (Shinwell *et al.*, 2003; Shinwell, 2005) is a higher order functional programming which supports the declaration of nominal datatypes very similar to those described in Definition 2.1. Object-level names are represented as permutative "atoms", which are generated as a side-effect of evaluation much as existential variables are generated in αML, with the difference that atoms in FreshML *only* represent object-level names and are each assumed to be globally unique. Object-level binders are deconstructed by *generative unbinding*, for which various correctness theorems have been proved (Shinwell & Pitts, 2005; Pitts & Shinwell, 2008). In αML, this form of abstraction elimination is implemented by generating new existential variables and using an equality constraint to get a handle on the components of the abstraction. FreshML therefore provides similar features to αML, but since there is no inbuilt notion of inductive definitions in the language, proof-search behaviours must be manually programmed whenever they are required. The key differences between FreshML and αML are that αML dispenses with the *permutative convention* of Gabbay & Mathijssen (2008), allowing any variables denoting names to be aliased or held distinct, and that αML combines functional and logic programming at the language level.

αProlog (Cheney & Urban, 2004) is a logic programming language which uses nominal abstract syntax to encode object-language binding structures. In principle, name-aliasing similar to that permissible in αML is possible in αProlog, although in this case backchaining using the efficient nominal unification (Urban *et al.*, 2004) algorithm implemented in the experimental system is known to give an incomplete search procedure. This incompleteness can be avoided by restricting oneself to a subset of αProlog composed of well-formed programs for which nominal unification produces a complete proof search (Urban & Cheney, 2005) (this property can be verified by nominal matching). This subset of αProlog suffices for many applications; nonetheless from a theoretical perspective it is desirable to find a complete proof search procedure. This is possible in αProlog by using the more complicated equivariant unification algorithm in place of nominal unification (Cheney & Urban, 2008; Cheney, 2010). However, the constraint problem used in αML is syntactically simpler than equivariant unification, and yet the two decision problems have been shown to be equivalent (Lakin, 2011). Hence, the αML language and implementation retain the power of equivariant unification while providing a simpler interface to the user.

6.2 Higher order abstract syntax and logical frameworks

Higher order abstract syntax (HOAS) is an alternative representation of object languages involving names and binders. HOAS systems use the typed λ-calculus as their

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

meta-language, and all object-level name-binding is translated into λ-abstractions. HOAS techniques have been used in practice in a number of logical frameworks such as λProlog (Nadathur & Miller, 1988), Twelf (Pfenning & Schürmann, 1999), Delphin (Poswolsky & Schürmann, 2009) and Beluga (Pientka & Dunfield, 2010). Such representations can be convenient, since capture-avoiding substitution comes "for free" from meta-level application using the β-rule of λ-calculus. However, this only provides a single notion of substitution – others, such as parallel substitution, must still be defined by hand.

In order to find a logic programming system on HOAS, one needs a unification algorithm to perform resolution. Higher order unification (Huet, 1975) unifies typed λ-terms up to $\alpha\beta\eta$-conversion, but this is undecidable (Goldfarb, 1981). However, a decidable subproblem exists, called higher order pattern unification (Miller, 1991). A higher order pattern is just a simply typed λ-term where every free variable z is applied to a sequence of distinct bound variables. Full β-reduction is not required to compute the normal forms of higher order patterns: let β_0-conversion be the restriction of β-conversion to redexes of the form $(\lambda x.t)\,x$. The restricted form of HOAS over higher order patterns and using higher order pattern unification to decide equality modulo $\alpha\beta_0\eta$-conversion is known as λ-tree syntax (Miller, 2000).

Pattern unification has been used in Teyjus (Nadathur & Mitchell, 1999; Qi, 2009), an implementation of λ*Prolog* (Nadathur & Miller, 1988). Nominal and higher order pattern unification have been shown to be equivalent (Cheney, 2005; Levy & Villaret, 2008), and proof-search over specifications encoded using the ∇-quantifier and nominal techniques were related by Gacek (2010). Other tools based on HOAS, such as Abella (Gacek *et al.*, 2009) and Bedwyr (Baelde *et al.*, 2007), support meta-level reasoning *about* inductive definitions. These tools use the ∇-quantifier (Miller & Tiu, 2005) to build generic proofs about terms with locally scoped binders.

6.3 Other abstract syntax encodings

De Bruijn indices (de Bruijn, 1972) are a well-established, nameless technique for representing terms with binding: bound variables are replaced by a natural number index which records the number of λ symbols in scope between the bound variable and its binding occurrence. For example, the term $\lambda f.\lambda x.\,f\,x$ is written as $\lambda.\lambda.10$. This approach has the attractive property that two α-equivalent terms have the same representation, making de Bruijn terms particularly amenable to manipulation by a computer (they are frequently used to represent binding in compiler intermediate languages). However, de Bruijn indices are less convenient for reasoning by humans.

Locally nameless representations (McKinna & Pollack, 1999; Aydemir *et al.*, 2008; Sato & Pollack, 2010; Pollack *et al.*, 2012) exist partway between nominal and nameless encodings of abstract syntax. These approaches employ two flavours of names – one for bound "local" names and another for free "global" names, and retain some of the advantages of both names and nameless representations, such as canonical representations of terms modulo α-equivalence. Indeed, the encoding of ground trees into αML, introduced in Section 2.5 and studied in depth in Lakin and Pitts (2012), is similar in spirit to locally nameless representations, relying on a fixed bijection to translate free names of the ground

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

tree into αML variables while hiding the variables which correspond to the abstracted names under $\exists$-binders in the αML meta-language.

6.4 Other functional logic programming languages

The most prominent of these multi-paradigm languages is Curry (Hanus, 1997); another is Mercury (Somogyi *et al.*, 1996). From our perspective, the major drawback of most existing functional logic languages is that they lack built-in support for programming with names and binders. A notable exception is Qu-Prolog (Nickolas & Robinson, 1996) which extends Prolog with features for quantifying object-language variables and explicit substitutions. The Qu-Prolog unification algorithm is semi-decidable, and is closer in spirit to equivariant unification (Cheney, 2010) than to nominal unification (Urban *et al.*, 2004).

The main problem encountered by functional logic languages is how to deal with the application of a function (which are typically defined by cases) to an unknown term, i.e. expressions like `f(X)`. There are two common solutions:

1. *Residuation* (Albert *et al.*, 2002) involves suspending the current computation in the hope that some other thread may compute an instantiation for `X`. The language must include concurrency primitives and the strategy is not guaranteed to succeed: if no instantiation is found, the computation fails by *floundering*.
2. *Narrowing* amounts to trying all possible (outermost) term constructors in the hope that one or more of these "guesses" may succeed. A strategy for performing narrowing only when absolutely necessary is described in Antoy *et al.* (2000).

In practice, Curry supports both residuation and narrowing, and the programmer can decide which to use in any given situation. αML only supports narrowing to avoid overcomplicating the language with additional concurrency primitives.

7 Discussion

We have defined a simple yet expressive notion class of α-inductive definitions, which involve names and binders, and presented a simple encoding of these into the αML meta-language as recursive functions. Our key technical result (Theorem 5.6) was that contextual equivalence between these recursive functions corresponds precisely to semantic equivalence between α-inductive definitions. This is a strong result which demonstrates that the αML meta-language correctly encodes proof-search computations over object-languages involving name-binding, since semantically equivalent terms relating to proof-search computations over inductively defined relations cannot be distinguished based on successful termination.

At first glance, the ability to bind a single name within a term seems quite restrictive compared with more expressive schemes such as those of `ott` (Sewell *et al.*, 2007) and Cαml (Pottier, 2006). This design decision was made for simplicity and also because one can get quite a long way using just single binding, particularly when coupled with the ability to assert arbitrary freshness constraints, as is the case in αML. To our knowledge, there are currently no concrete mathematical results on the relative expressive power of

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

```
       sub(t,z,t',t'')
------------------------------ [beta_sub where z:var, t,t',t'':lam]
 beta(App((Lam <z>t),t'), t'')

       beta(t,t'')
------------------------------ [beta_app where t,t',t'':lam]
 beta(App(t,t'), App(t'',t'))
```

Fig. 12. Example of ASCII syntax for α-inductive definitions in αML, defining the `beta` relation from Example 3.11 in terms of the `sub` relation (rules not shown).

different binding specification languages in the literature, although a number of projects do offer comparative case studies (Aydemir *et al.*, 2005; Weirich *et al.*, 2011).

A prototype implementation of the αML programming language is available for download from the first author's webpage. The interpreter implements the operational semantics from Figure 3, using the constraint-solving algorithm from Lakin (2011) to decide satisfiability of constraint problems. The implementation supports syntactic sugar for αML expressions which encode ground trees, as described in Lakin & Pitts (2012), and for encoding α-inductive definitions, as described above. Figure 12 gives an example of the ASCII syntax for defining the `beta` relation from Example 3.11 in the αML interpreter. Assuming that the `sub` relation is also defined, this code is automatically translated into the corresponding recursive function (which was presented in Example 4.1).

The correctness results proved in this paper open up new possibilities for compile-time transformations of α-inductive definitions prior to their translation into the corresponding αML function, which could allow for more efficient execution of user queries. If we can show that the transformation preserves the semantics of the definition, then the results from this paper tell us that the resulting αML implementations are contextually equivalent and hence could not be distinguished based on their successful termination behaviour. As a simple example, a user-supplied relation which axiomatizes α-equivalence at a particular type could be replaced with an appropriate instance of the equality constraint primitive of the αML meta-language.

Since αML incorporates both functional and CLP features, we may consider rewriting α-inductive definitions to include function evaluations within the proof-search procedure. For example, in the α-inductive definition from Example 3.11, the `sub` relation may be considered "functional" in the sense that if $\texttt{sub}(t,z,t',t'')$ holds then the "output" t'' is determined by the "inputs" t, z and t'. The development of a mode system for αML, similar to those developed for Prolog and Curry (Hanus & Zartmann, 1994), would allow us to detect and rewrite such expressions at compile-time if we could prove a contextual equivalence result for the relational and functional versions of the computation. The fact that both functional and CLP styles of programming are available in αML means that the programmer is free to choose the style most suited to the problem at hand. While this is largely a matter of personal taste, inductively defined relations, such as type inference and reduction relations, are examples of programs that can be expressed very naturally in the CLP fragment of αML, as shown in Example 3.11. On the other hand, recursively defined functions, such as normalization by evaluation and closure conversion, may be written

https://doi.org/10.1017/S0956796813000245 Published online by Cambridge University Press

more straightforwardly using just the functional features of αML, which are not the focus of this paper.

A notable omission from the grammar of schematic formulae in Definition 3.3 is a negation formula $\neg\varphi$, with the semantics that $\neg\varphi$ is satisfied precisely when φ is not satisfied. If negative occurrences of atomic formulae were permitted, then the monotonicity property of the semantics of schematic formulae, to which we appealed in Definition 3.10 to deduce the existence of a least fixed point in the semantics of α-inductive definitions, would no longer hold. From a practical perspective, under the standard "negation as failure" interpretation of negation, the implementation of a negation formula could not be complete in general, as if the evaluation of φ diverges then the evaluation of $\neg\varphi$ should terminate successfully. For these reasons if negation-like behaviour is required in an αML program, the user must currently directly implement a relation with the desired semantics. We saw this in Example 3.11 with the `nf` predicate, which was defined to hold for precisely those terms for which no further reduction steps could be derived using the `beta` relation.

Previous work on non-determinism checking and analysis (Hanus & Steiner, 1998; Hanus & Steiner, 2000; Braßel & Hanus, 2005) for the functional logic language Curry (Hanus & Zartmann, 1994) could be fruitfully applied to αML. This would be beneficial because the evaluation of a branching expression creates multiple non-deterministic branches of the configuration, but there is no way to subsequently bring these branches back together and continue the computation in a deterministic way. Incorporating language-level features for delimiting the spread of non-determinism through the computation would enable the abstract machine to control the search process, for example by cancelling unwanted branches when enough solutions have already been found. This addition to the language would make it more convenient to incorporate computations over inductively defined relations into traditional functional programs.

Finally, we note that the properties of the contextual equivalence relation presented above depend on the operational behaviours of αML expressions which we permit ourselves to observe. In particular, if we allow finite failure (Definition 2.8) to be observed as well as successful termination, we obtain a more fine-grained equivalence relation which we will call $\cong^{\circ}_{F}$. This relation has many of the same advantageous properties as $\cong^{\circ}$, however, the theorem corresponding to Theorem 5.5 does not hold if we use $\cong^{\circ}_{F}$ instead of $\cong^{\circ}$. This is because $\cong^{\circ}_{F}$ allows us to distinguish between formulae which diverge and those which fail finitely, which are both unsatisfiable in the semantics of schematic formulae presented in Section 3.3. Such formulae cannot be distinguished using $\cong^{\circ}$ since only successful termination is observable. We refer the interested reader to Section 5.6 of Lakin (2010) for further discussion of this point, but note that we still consider the $\cong^{\circ}$ relation studied here to be a reasonable notion of equivalence because observing just successful termination is standard in program equivalence studies for traditional functional programming languages.

Acknowledgments

The authors thank Paul Blain Levy and anonymous reviewers for help debugging the proofs, and anonymous reviewers for suggesting other improvements to the manuscript.

References

Albert, E., Hanus, M., Huch, F., Oliver, J. & Vidal, G. (2002) An operational semantics for declarative multi-paradigm languages. In *Proceedings of the 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, Comini, M. & Falaschi, M. (eds), Electronic Notes in Theoretical Computer Science, vol. 76. Philadelphia PA: Elsevier, pp. 1–19.

Antoy, S., Echahed, R. & Hanus, M. (2000) A needed narrowing strategy. *J. ACM* **47**(4), 776–822.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, Necula, G. C. & Wadler, P. (eds). New York, NY: ACM Press, pp. 3–15.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005) Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Hurd, J. & Melham, T. F. (eds), Lecture Notes in Computer Science, vol. 3603. Berlin, Germany: Springer-Verlag, pp. 50–65.

Baelde, D., Gacek, A., Miller, D., Nadathur, G. & Tiu, A. (2007) The Bedwyr system for model checking over syntactic expressions. In *Proceedings of the 21st International Conference on Automated Deduction (CADE 2007)*, Pfenning, F. (ed), Lecture Notes in Computer Science, vol. 4603. Berlin, Germany: Springer-Verlag, pp. 391–397.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, Revised edn. Amsterdam, Netherlands: North-Holland.

Braßel, B. & Hanus, M. (2005) Nondeterminism analysis of functional logic programs. In *Proceedings of the 21st International Conference on Logic Programming (ICLP 2005)*, Gabbrielli, M. & Gupta, G. (eds), Lecture Notes in Computer Science, vol. 3668. Berlin, Germany: Springer-Verlag, pp. 265–279.

Cheney, J. (2005) Relating nominal and higher order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*, Vigneron, L. (ed), LORIA research report A05-R-022, pp. 104–119.

Cheney, J. (2010) Equivariant unification. *J. Autom. Reasoning* **45**(3), 267–300.

Cheney, J. & Urban, C. (2004) Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, Demoen, B. & Lifschitz, V. (eds), Lecture Notes in Computer Science, no. 3132. Berlin, Germany: Springer-Verlag, pp. 269–283.

Cheney, J. & Urban, C. (2008) Nominal logic programming. *ACM Trans. Program. Lang. Syst.* **30**(5), 1–47.

Clark, K. L. (1978) Negation as failure. In *Logic and Data Bases*, Gallaire, J. & Minker, J. (eds). New York, NY: Plenum Press, pp. 293–322.

de Bruijn, N. (1972) Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church–Rosser Theorem. *IIndag. Math.* **34**, 381–392.

Dershowitz, N. & Manna, Z. (1979) Proving termination with multiset orderings. *Commun. ACM* **22**(8), 465–476.

Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with continuations. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1993). *ACM SIGPLAN Not.* **28**(6), 237–247 (New York, NY: ACM Press).

Gabbay, M. J. & Mathijssen, A. (2008) Capture-avoiding substitution as a nominal algebra. *Form. Asp. Comput.* **20**(4–5), 451–479.

Gabbay, M. J. & Pitts, A. M. (2002) A new approach to abstract syntax with variable binding. *Form. Asp. Comput.*, **13**(3–5), 341–363.

Gacek, A. (2010) Relating nominal and higher order abstract syntax specifications. In *PPDP '10: Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*. New York, NY: ACM, pp. 177–186.

Gacek, A., Miller, D. & Nadathur, G. (2009) Reasoning in Abella about structural operational semantics specifications. In *Proceedings of the 3rd International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, Abel, A. & Urban, C. (eds), Electronic Notes in Theoretical Computer Science, vol. 228. Philadelphia PA: Elsevier, pp. 85–100.

Goldfarb, W. D. (1981) The undecidability of the second-order unification problem. *Theor. Comput. Sci.* **13**(2), 225–230.

Gordon, A. D. (1998) *Operational Equivalences for Untyped and Polymorphic Object Calculi*. Cambridge, UK: Newton Institute, Cambridge University Press.

Hanus, M. (1997) A unified computation model for declarative programming. In *Proceedings of the 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE 1997)*, Falaschi, M., Navarro, M. & Policriti, A. (eds), pp. 9–24.

Hanus, M. (2007) Multi-paradigm declarative languages. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007)*, Dahl, V. & Niemelä, I. (eds), Lecture Notes in Computer Science, vol. 4670. Berlin, Germany: Springer-Verlag, pp. 45–75.

Hanus, M. & Steiner, F. (1998) Controlling search in declarative programs. In *Principles of Declarative Programming (Proceedings of the Joint International Symposium PLILP/ALP 1998)*, Goos, G., Hartmanis, J. & van Leeuwen, J. (eds), Lecture Notes in Computer Science, vol. 1490. Springer-Verlag, pp. 374–390.

Hanus, M. & Steiner, F. (2000) Type-based nondeterminism checking in functional logic programs. In *Proceedings of the 2nd International ACM-SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, New York, NY: ACM Press, pp. 202–213.

Hanus, M. & Zartmann, F. (1994) Mode analysis of functional logic programs. In *Proceedings of the 1st International Static Analysis Symposium (SAS 1994)*, le Charlier, B. (ed), Lecture Notes in Computer Science, vol. 864. Berlin, Germany: Springer-Verlag, pp. 26–42.

Howe, D. J. (1996) Proving congruence of bisimulation in functional programming languages. *Inf. Comput.* **124**(2), 103–112.

Huet, G. (1975) A unification algorithm for typed λ-calculus. *Theor. Comput. Sci.* **1**(1), 27–57.

Jaffar, J., Maher, M., Marriott, K. & Stuckey, P. (1998) Semantics of constraint logic programming. *J. Log. Program.* **37**(1–3), 1–46.

Lakin, M. R. (2010) *An Executable Meta-language for Inductive Definitions with Binders*, PhD thesis, University of Cambridge, UK.

Lakin, M. R. (2011) Constraint-solving in non-permutative nominal abstract syntax. *Logical Methods Comput. Sci.* **7**(3:06), 1–31.

Lakin, M. R. & Pitts, A. M. (2009) Resolving inductive definitions with binders in higher-order typed functional programming. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*, Castagna, G. (ed), Lecture Notes in Computer Science, vol. 5502. Berlin, Germany: Springer-Verlag, pp. 47–61.

Lakin, M. R. & Pitts, A. M. (2012) Encoding abstract syntax without fresh names. *J. Autom. Reasoning* **49**(2), 115–140.

Lassen, S. B. (1998) *Relational Reasoning about Contexts*. Cambridge, UK: Newton Institute, Cambridge University Press.

Levy, J. & Villaret, M. (2008) Nominal unification from a higher-order perspective. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, Voronkov, A. (ed), Lecture Notes in Computer Science, vol. 5117. Berlin, Germany: Springer-Verlag, pp. 246–260.

Mason, I. A. & Talcott, C. L. (1991) Equivalence in functional languages with effects. *J. Funct. Program.* **1**(3), 287–327.

McKinna, J. & Pollack, R. (1999) Some lambda calculus and type theory formalized. *J. Autom. Reason.* **23**(3), 373–409.

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

Miller, D. (2000) Abstract syntax for variable binders: An overview. In *Proceedings of Computational Logic – CL 2000, First International Conference,* London, UK, 24–28 July 2000, Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L. Moniz, Sagiv, Y. & Stuckey, P. J. (eds), Lecture Notes in Computer Science, vol. 1861. Berlin, Germany: Springer-Verlag, pp. 239–253.

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

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML* (revised). Cambridge, MA: MIT Press.

Nadathur, G. & Miller, D. (1988) An overview of λProlog. In *Proceedings of the 5th International Conference on Logic Programming (ICLP 1988)*, Kowalski, R. A. & Bowen, K. A. (eds). Cambridge, MA: MIT Press, pp. 810–827.

Nadathur, G. & Mitchell, D. J. (1999) System description: Teyjus – a compiler and abstract machine based implementation of λProlog. In *Automated Deduction—Cade-16*, Ganzinger, H. (ed), Lecture Notes in Computer Science, vol. 1632. Berlin, Germany: Springer-Verlag, pp. 287–291.

Nickolas, P. & Robinson, P. J. (1996) The Qu-Prolog unification algorithm: Formalisation and correctness. *Theor. Comput. Sci.* **169**(1), 81–112.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE 1999)*, Ganzinger, H. (ed), Lecture Notes in Artifical Intelligence, vol. 1632. Berlin, Germany: Springer-Verlag, pp. 202–206.

Pientka, B. & Dunfield, J. (2010) Beluga: A framework for programming and reasoning with deductive systems (System Description). In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, Edinburgh, UK, 16–19 July 2010, Giesl, J. & Hähnle, R. (eds), Lecture Notes in Computer Science, vol. 6173. Berlin, Germany: Springer-Verlag, pp. 15–21.

Pitts, A. M. (2002) Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, Lecture Notes in Computer Science, vol. 2395. Berlin, Germany: Springer-Verlag, pp. 378–412.

Pitts, A. M. (2003) Nominal logic, a first-order theory of names and binding. *Inf. Comput.* **186**(2), 165–193.

Pitts, A. M. (2005) Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed). Cambridge, MA: MIT Press, Chap. 7, pp. 245–289.

Pitts, A. M. (2006) Alpha-structural recursion and induction. *J. ACM* **53**(3), 459–506.

Pitts, A. M. (2011) Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, Sangiorgi, D. & Rutten, J. (eds), Cambridge Tracts in Theoretical Computer Science, vol. 52. Cambridge, UK: Cambridge University Press, Chap. 5, pp. 197–232.

Pitts, A. M. & Shinwell, M. R. (2008) Generative unbinding of names. *Logical Methods Comput. Sci.* **4**(1:4), 1–33.

Pollack, R., Sato, M. & Ricciotti, W. (2012) A canonical locally named representation of binding. *J. Autom. Reasoning* **49**(2), 185–207.

Poswolsky, A. & Schürmann, C. (2009) System Description: Delphin—a functional programming language for deductive systems. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008). *Electron. Notes Theor. Comput. Science*, **228**, 113–120.

Pottier, F. (2006) An overview of Cαml. In *Proceedings of the 2005 ACM-SIGPLAN Workshop on ML (ML 2005)*, Benton, N. & Leroy, X. (eds). Philadelphia PA: Elsevier, pp. 27–52. (*Electron. Notes Theor. Comput. Sci.* **148**(2)).

Qi, X. (2009) *An Implementation of the Language λProlog Organized Around Higher-order Pattern Unification*. PhD thesis, University of Minnesota.

Sato, M. & Pollack, R. (2010) External and internal syntax of the lambda-calculus. *J. Symb. Comput.* **45**, 598–616.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strniša, R. (2007) Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, Hinze, R. & Ramsey, N. (eds). New York, NY: ACM Press, pp. 1–12.

Shinwell, M. R. (2005) *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, University of Cambridge, UK.

Shinwell, M. R. & Pitts, A. M. (2005) On a monadic semantics for freshness. *Theor. Comput. Sci.* **342**(1), 28–55.

Shinwell, M. R., Pitts, A. M. & Gabbay, M. J. (2003) FreshML: Programming with binders made simple. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, Runciman, C. & Shivers, O. (eds). New York, NY: ACM Press, pp. 263–274.

Somogyi, Z., Henderson, F. & Conway, T. (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Funct. Program.* **29**(1–3), 17–64.

Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.* **5**(2), 285–309.

Urban, C. & Cheney, J. (2005) Avoiding equivariance in Alpha-Prolog. In *the Proceedings of the 7th International Conference on Typed Lambda Calculus and Applications (TLCA 2005)*, Urzyczyn, P. (ed), Lecture Notes in Computer Science, no. 3461. Berlin, Germany: Springer-Verlag, pp. 74–89.

Urban, C., Pitts, A. M. & Gabbay, M. J. (2004) Nominal unification. *Theor. Comput. Sci.* **323**(1–3), 473–497.

Weirich, S., Yorgey, B. A. & Sheard, T. (2011) Binders unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, Tokyo, Japan, 19–21 September 2011, Chakravarty, M. M. T., Hu, Z. & Danvy, O. (eds), pp. 333–345.