Resolving Inductive Definitions with Binders in Higher-Order Typed Functional Programming*

Matthew R. Lakin and Andrew M. Pitts

University of Cambridge Computer Laboratory, Cambridge CB3 0FD, UK
{Matthew.Lakin,Andrew.Pitts}@cl.cam.ac.uk

Abstract. This paper studies inductive definitions involving binders, in which aliasing between free and bound names is permitted. Such aliasing occurs in informal specifications of operational semantics, but is excluded by the common representation of binding as meta-level λ-abstraction. Drawing upon ideas from functional logic programming, we represent such definitions with aliasing as recursively defined functions in a higher-order typed functional programming language that extends core ML with types for name-binding, a type of "semi-decidable propositions" and existential quantification for types with decidable equality. We show that the representation is sound and complete with respect to the language's operational semantics, which combines the use of evaluation contexts with constraint programming. We also give a new and simple proof that the associated constraint problem is NP-complete.

1 Introduction

Perhaps the single most important technique in the study of programming language semantics is the use of inductive definitions. This is especially the case for operational semantics, which in broad terms consists of one or more inductively defined relations between data structures involving programming language syntax. The inductive definition commonly takes the form of finitely many "schematic" rules containing parameters that can be instantiated, usually in infinitely many different ways, to get concrete rules for inductively generating instances of the relations. Schematic rules are necessarily written in some meta-language whose definition is often left implicit in published research. Having to be completely precise about the meta-language of rule schemes is an inescapable part of the current trend toward mechanization of semantics, whether it be machine-assisted proof construction/checking, or executable semantic specifications. In this paper we are concerned with the latter, but in either case it is clear that the ubiquitous presence of binding constructs in the "object-language" (that is, the programming language whose semantics is being formalized) creates difficulties for mechanized meta-languages (see the *POPLmark Challenge* wiki, for example). Ideally one would like the executable meta-language for rule schemes to provide a fully automatic treatment of α-conversion of bound names

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

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 47–61, 2009.

in object-languages. Here we investigate a way of doing that within the context of higher-order typed functional programming. In doing so we take a "nominal" approach to object-level binders for the following reasons to do with *name aliasing*.

One way of dealing with issues of α-conversion is to make the representation of object-level binders completely anonymous. This can be achieved through a case-by-case use of de Bruijn indices [8], or more systematically by use of Higher-Order Abstract Syntax (HOAS) [21] to enforce that object-level bound names are only represented by meta-level bound names. In either case the conceptually simple operation of instantiating parameters in a rule scheme, which may involve capture of a free name by a binder of the same name, has to be replaced by something more complicated—simply because "binder of the same name" makes no sense if binders have been anonymized. But there is a more serious problem with anonymous representations of object-level binding: they rule out the common practice of *name-aliasing* involving binders, be it the use of the same name at two different binding occurrences, or the use of the same name for both a free and a binding occurrence. For example, in an inductive definition of β-reduction of λ-terms it is natural to use the rule scheme

$$\frac{t \to t'}{\lambda x.\, t \to \lambda x.\, t'}$$

where the two different binding occurrences within the conclusion are both named x. In this case the name-aliasing does not cause a problem for a formalization using HOAS, which might render the above rule as

$$\frac{f(x) \to f'(x)}{\lambda(f) \to \lambda(f')}$$

where f and f' are meta-variables of function type. The top half of Fig. 1 contains another example. (The bottom half of the figure will be explained in Sect. 2.) In this case the conclusion of the third rule contains both free (the first x) and binding (the x within $\mathtt{L}(\langle x \rangle t)$) occurrences with the same name. We leave the reader to ponder how to convert these rules into an extensionally equivalent HOAS formalization (probably by ignoring the third rule completely). In fact we do not know any definitive results comparing the class of relations (on tuples of α-equivalence classes of λ-terms, say) defined by HOAS rule schemes with the class defined by first-order rule schemes with conventional, named binders. In any case, the phenomenon of name-aliasing seems too convenient to give up unless we really have to.

So we advocate the study of executable meta-languages for rule schemes that allow object-level binders to be named. More specifically we study such an executable meta-language which is a higher-order, typed functional programming language, drawing upon the ideas of *functional logic programming* [14]. Our motivation for favouring this paradigm over the relational paradigm of first-order logic programming has to do with the expressiveness and modularity afforded by

$\mathtt{nfv}(x,t)$: "$x : \mathtt{vr}$ is not a free variable of the λ-term $t : \mathtt{tm}$"

$$\frac{x \,\#\, x'}{\mathtt{nfv}(x, \mathtt{V}\, x')} \qquad \frac{\mathtt{nfv}(x,t)\ \&\ \mathtt{nfv}(x,t')}{\mathtt{nfv}(x, \mathtt{A}(t,t'))} \qquad \frac{}{\mathtt{nfv}(x, \mathtt{L}(\texttt{<}x\texttt{>}t))} \qquad \frac{\mathtt{nfv}(x,t)\ \&\ x \,\#\, x'}{\mathtt{nfv}(x, \mathtt{L}(\texttt{<}x'\texttt{>}t))}$$

Equivalent standard form $\dfrac{\varphi}{\mathtt{nfv}\, y}$ has

$$\varphi \triangleq \exists x, x'(y = (x, \mathtt{V}\, x')\ \&\ x \,\#\, x') \ \vee\ \exists x, t, t'(y = (x, \mathtt{A}(t,t'))\ \&\ \mathtt{nfv}(x,t)\ \&\ \mathtt{nfv}(x,t')) \ \vee\ \exists x, t(y = \mathtt{nfv}(x, \mathtt{L}(\texttt{<}x\texttt{>}t))) \ \vee\ \exists x, x', t(y = \mathtt{nfv}(x, \mathtt{L}(\texttt{<}x'\texttt{>}t))\ \&\ \mathtt{nfv}(x,t)\ \&\ x \,\#\, x')$$

Fig. 1. Example α-inductive definition

higher-order functions. For example, having higher-order functions allows one to encode definitions that are parameterised by other definitions (such as the various operations on relations that occur in the relational approach to contextual equivalence of programs [12, 17, 25]).

Contributions of this paper. We begin by fixing a simple, yet expressive class of inductive definitions permitting name-aliasing in binders, where binding is handled generically through the existing notion of a "nominal signature". These *α-inductive definitions* (Sect. 2) may involve side-conditions asserting constraints in terms of α-equivalence and the "not-a-free-variable-of" relation. In Sect. 3 we make the apparently new observation that such constraints can express membership in finite sets; consequently the associated constraint satisfaction problem is NP-complete (Theorem 3.3). Section 4 introduces the main contribution of the paper, a typed higher-order functional programming language that extends core ML with name-binding types, a type of "semi-decidable propositions" and existential quantification for types in a class of equality types coinciding with the arities of a user-declared nominal signature. This language, which we call αML, draws upon the ideas of αProlog [7], extending them to higher-order functional programming. αML is a simplification of both our first attempt to do this [16] and of αProlog itself, in that *it avoids the use of concrete names and name-permutations in programs* (see Remark 2.2 and Sect. 6 for the significance of this). αML has a remarkably simple operational semantics that combines the use of Felleisen-style evaluation contexts with constraint programming; we show that it restricts to the usual operational semantics on the purely functional part of αML (Theorem 4.1). By design, αML represents α-inductive definitions as certain recursively defined functions; we prove that this representation is sound and complete (Corollary 5.3). Finally, Sect. 6 discusses related and future work.

2 α-Inductive Definitions

In this section we give a simple, yet expressive class of *inductively defined relations between α-equivalence classes of expressions*, or *α-inductive definitions* for short. Alpha-equivalence arises from the presence of binding constructs in

the expressions and we will deal with this in a generic way by using *nominal signatures* [28], Σ. These generalize the usual notion of many-sorted algebraic signature to encompass constructors that bind names of various sorts. Such a Σ is given by a finite set of *name sorts* N, a disjoint finite set of *data sorts* S, and a finite set of *constructors* $K : A \to S$, each with a specified result sort S and argument arity A—where the *nominal arities* of the signature are given by:

$$\begin{array}{lll} A ::= & S & \text{(data sort)} \\ & A * \cdots * A & \text{(tuples)} \\ & N & \text{(name sort)} \\ & [N]A & \text{(name-abstractions).} \end{array} \tag{1}$$

For example, the nominal signature for untyped λ-calculus has a name sort `vr` for variables, a data sort `tm` for λ-terms and constructors `V : vr -> tm`, `A : tm*tm -> tm` and `L : [vr]tm -> tm`. For other examples, see [24, Sect. 2]. As in that paper, we associate with each arity A of a nominal signature Σ a set $\alpha\text{-}Tree_\Sigma(A)$ of *α-equivalence classes of abstract syntax trees*, or *α-trees* for short. The elements of each $\alpha\text{-}Tree_\Sigma(A)$ are equivalence classes $[g]_\alpha$ of syntax trees $g \in Tree_\Sigma(A)$ built up from countably many *names* $n \in Name(N)$ (for each name sort N of Σ) by repeatedly applying the following three operations.

Constructor application: $K\, g \in Tree_\Sigma(S)$, if $g \in Tree_\Sigma(A)$ and $K : A \to S$.
Tupling: $(g_1, \ldots, g_n) \in Tree_\Sigma(A_1 * \cdots * A_n)$, if $g_i \in Tree_\Sigma(A_i)$ for $i = 1..n$.
Name-abstraction: $\langle n \rangle g \in Tree_\Sigma([N]A)$, if $n \in Name(N)$ and $g \in Tree_\Sigma(A)$.

(These trees are the *ground* nominal terms from [28], that is, the ones not involving variables.) The third operation, name-abstraction, is the generic binding form provided by nominal signatures: renaming $\langle n \rangle(-)$-bound names in trees gives an equivalence relation $=_\alpha$ [24, Fig. 1] and $\alpha\text{-}Tree_\Sigma(A)$ is the quotient $Tree_\Sigma(A)/=_\alpha$. To specify inductively defined relations between α-trees we make use of a simple meta-language of patterns.

Definition 2.1 (patterns and valuations). The *patterns* $p \in Pat_\Sigma(A)$ for describing α-trees of each arity A of Σ are built up from countably many *variables* $x \in Var(A)$ (for each A) by repeatedly applying the three tree-forming operations mentioned above:
Constructor application: $K\, p \in Pat_\Sigma(S)$, if $p \in Pat_\Sigma(A)$ and $K : A \to S$.
Tupling: $(p_1, \ldots, p_n) \in Pat_\Sigma(A_1 * \cdots * A_n)$, if $p_i \in Pat_\Sigma(A_i)$ for $i = 1..n$.
Name-abstraction: $\langle x \rangle p \in Pat_\Sigma([N]A)$, if $x \in Var(N)$ and $p \in Pat_\Sigma(A)$.

A *valuation* V is a finite function mapping variables to α-trees (of the same arity). If the variables occurring in a pattern $p \in Pat_\Sigma(A)$ are in $dom(V)$ (the domain of definition of V), then $[\![p]\!]_V \in \alpha\text{-}Tree_\Sigma(A)$ denotes the α-tree resulting from p by replacing each $x \in dom(V)$ with $V(x)$.

Remark 2.2. The following points about patterns should be noted.

(a) *Variables stand for unknown α-trees, not unknown trees, and (hence) a pattern $p \in Pat_\Sigma(A)$ describes an α-tree rather than a tree* (just which one depends upon how its variables are instantiated by a valuation). This reflects the common

practice of leaving α-equivalence implicit and referring to a class via a representative, signalled by a phrase like "we identify expressions up to α-equivalence". (Our own Figs. 2 and 3 in Sect. 4 provide examples of this!)

(b) *No concrete names $n \in Name(N)$ occur in patterns.* In particular, although the meta-language allows us to name object-level binding occurrences, $\texttt{<}x\texttt{>}p$, we use variables x of name sort rather than names themselves to do so. Again, this reflects common practice. For example, in Barendregt's classic text [1], Definition 2.1.1 says that λ-terms are words over an alphabet containing, among other things, "variables $v_0, v_1, \ldots$"; then Notation 2.1.2 says that "$x, y, z \ldots$ denote arbitrary variables"; the concrete variables $v_0, v_1, \ldots$ are never mentioned again and only the meta-variables $x, y, z \ldots$ are used throughout the rest of the book.

(c) *There are no meta-level variable-binding constructs in patterns—all variables in a pattern are free.* In particular x occurs free in the name-abstraction pattern $\texttt{<}x\texttt{>}p$. This allows patterns to support the phenomenon of name-aliasing discussed in the Introduction.

(d) *Valuation of patterns is a form of "possibly-capturing" substitution.* Once again, this reflects common practice when instantiating the meta-variables of schematic rules in operational semantics. Note that, in addition to the previous point, this is another reason why it makes no sense to try to identify patterns up to renaming $\texttt{<}x\texttt{>}(-)$-scoped variables, since valuations do not respect such a notion of α-equivalence. For example, we cannot regard $\texttt{<}x\texttt{>}z$ and $\texttt{<}y\texttt{>}z$ as equivalent (where x, y and z are distinct), since the valuation $V = \{x \mapsto [n]_\alpha, y \mapsto [n']_\alpha, z \mapsto [n]_\alpha\}$ (with $n \neq n'$) has $[\![\texttt{<}x\texttt{>}z]\!]_V = [\langle n \rangle n]_\alpha \neq [\langle n' \rangle n]_\alpha = [\![\texttt{<}y\texttt{>}z]\!]_V$.

Fix a finite set of *relation symbols* $r <: A$, each with a specified arity A. Such an r is intended to denote a subset of $\alpha\text{-}Tree_\Sigma(A)$. Schematic rules for inductively defining such subsets take the form

$$\frac{r_1(p_1)\ \&\ \cdots\ \&\ r_m(p_m)\ \&\ c_1\ \&\ \cdots\ \&\ c_n}{r(p)} \tag{2}$$

(see Fig. 1 for an example). The conclusion of (2) is an *atomic formula* $r(p)$ with $r <: A$ and $p \in Pat_\Sigma(A)$ for some arity A; the hypothesis is a finite (possibly empty) conjunction of such atomic formulas and "side-conditions" c_i, that is, constraints on how the rule may be instantiated by a valuation. What form of constraints should we use? At the very least we need *name-inequality* constraints $x \neq x'$, where $x, x' \in Var(N)$ with N a name sort. Experience with nominal logic [23, 11] and nominal logic programming [7] shows that it is useful to generalize name-inequality to *name freshness* constraints, $x \,\#\, p$ where $x \in Var(N)$ and $p \in Pat_\Sigma(A)$, even though these can be inductively defined in terms of name-inequality (cf. Fig. 1). The intended meaning of $x \,\#\, p$ on α-trees is as follows.

Definition 2.3 (free names and freshness). If $t \in \alpha\text{-}Tree_\Sigma(A)$ we write $FN\,t$ for the finite set of names that occur freely in some (indeed, any) tree g with $[g]_\alpha = t$; in other words, $n \in FN[g]_\alpha$ iff n occurs in g, but not within the

scope of any name-abstraction $\langle n \rangle(-)$. Note that each α-tree $t \in \alpha\text{-}Tree_\Sigma(N)$ of name sort N is of the form $t = [n]_\alpha = \{n\}$ for some $n \in Name(N)$ (because the constructors of a nominal signature only produce results of data sort, rather than of more general arities, and these are disjoint from the name sorts). Given $t \in \alpha\text{-}Tree_\Sigma(N)$ and $t' \in \alpha\text{-}Tree_\Sigma(A)$, we write $t \# t'$ and say t *is fresh for* t' if $t = [n]_\alpha$ and $n \notin FN\, t'$.

Note that the case of several mutually inductively defined relation symbols $r_1 <: A_1, \ldots, r_k <: A_k$ reduces to the case of a single one at the expense of extending the signature: we add a new data sort S and new constructors $I_i : A_i \to S$ for $i \in \{1..k\}$ and use the fact that subsets $R \subseteq \alpha\text{-}Tree_\Sigma(S)$ are in bijection with n-tuples of subsets, $R_1 \subseteq \alpha\text{-}Tree_\Sigma(A_1), \ldots, R_n \subseteq \alpha\text{-}Tree_\Sigma(A_n)$. So from now on *we will fix on a single arity* A_r *of a nominal signature* Σ *and a single relation symbol* $r <: A_r$.

Definition 2.4 (formulas and satisfaction). Let $Form_\Sigma$ be the set of first-order formulas built up from atomic formulas $r(p)$ (where $p \in Pat_\Sigma(A_r)$), equalities $p = p'$ ($p, p' \in Pat_\Sigma(A)$, some A) and freshnesses $x \# p$ ($x \in Var(N)$, $p \in Pat_\Sigma(A)$, some N, A) just using finite conjunctions $\&$, finite disjunctions $\vee$ and existential quantification $\exists x(-)$ ($x \in Var(A)$, some A). Given an interpretation $R \subseteq \alpha\text{-}Tree_\Sigma(A_r)$ for the relation symbol r and a valuation V (Definition 2.1) for the free variables of $\varphi \in Form_\Sigma$ (i.e. those not within the scope of an $\exists x$), let $(R, V) \models \varphi$ denote the associated *satisfaction relation*. Thus $(R, V) \models r(p)$ holds if $[\![p]\!]_V \in R$; $(R, V) \models p = p'$ holds if $[\![p]\!]_V = [\![p']\!]_V$; $(R, V) \models x \# p$ holds if $V(x) \# [\![p]\!]_V$ (Definition 2.3); and satisfaction is extended to compound formulas in the usual way.

Generalizing (2), we will allow the hypothesis of a schematic rule to be a formula in $Form_\Sigma$. Allowing equality, disjunction and existential quantification in addition to freshness and conjunction does not increase the expressive power of inductive definitions; but it does allow us to write inductive definitions in a "standard form", illustrated in Fig. 1.

Definition 2.5 (standard α-inductive definitions). An α*-inductive definition* $\mathcal{D}$ *in standard form* of a set of α-trees of arity A_r is given by

$$\frac{\varphi}{r(x)} \tag{3}$$

where $x \in Var(A_r)$ and $\varphi \in Form_\Sigma$ is a formula with at most x free. The *meaning* $[\![\mathcal{D}]\!] \subseteq \alpha\text{-}Tree_\Sigma(A_r)$ of $\mathcal{D}$ is by definition the least fixed point of the monotone function $\Phi_\mathcal{D}$ on subsets of α-trees that maps each $R \subseteq \alpha\text{-}Tree_\Sigma(A_r)$ to

$$\Phi_\mathcal{D}(R) \triangleq \{t \in \alpha\text{-}Tree_\Sigma(A_r) \mid (R, \{x \mapsto t\}) \models \varphi\}. \tag{4}$$

The definition of $[\![\mathcal{D}]\!]$ via (4) is a fancy way of stating the usual meaning of a rule-based inductive definition: the rule (3) is schematic in the sense that it has the variable x as parameter; we instantiate x to get many concrete rules (this

is the effect of the valuations $\{x \mapsto t\}$ in the definition of $\Phi_{\mathcal{D}}$) and take the least set of α-trees closed under these rules, in other words, the least R such that $\Phi_{\mathcal{D}}(R) \subseteq R$. The existence of $[\![\mathcal{D}]\!]$ is an application of the usual Tarski fixed point theorem ($\Phi_{\mathcal{D}}$ is monotone because the relation symbol r only occurs positively in φ). Indeed $\Phi_{\mathcal{D}}$ is finitary and we can construct $[\![\mathcal{D}]\!]$ as the union of the countable chain of subsets $\emptyset \subseteq \Phi_{\mathcal{D}}(\emptyset) \subseteq \Phi_{\mathcal{D}}(\Phi_{\mathcal{D}}(\emptyset)) \subseteq \cdots$ of $\alpha\text{-}Tree_{\Sigma}(A_r)$.

3 α-Tree Constraint Problems

The hypothesis φ of an α-inductive definition (3) contains non-inductive equality and freshness constraints. When instantiated by a particular valuation, the validity of such constraints amounts to α-equivalence of trees $g \in Tree_{\Sigma}(A)$ and to non-membership of the set of free names of such trees. These are properties that can be decided in linear time; see [3] for example. However, the problem of checking whether or not there is some valuation that validates a collection of equality and freshness constraints is surprisingly more complicated, mainly because of the presence of variables x in binding position in name-abstraction patterns $\langle x \rangle p$ (see points (c) and (d) in Remark 2.2).

Definition 3.1 (constraints and their satisfaction). A formula $\varphi \in Form_{\Sigma}$ is an α-tree *constraint* if it is of the form $\exists x_1 \cdots \exists x_m\, (c_1 \;\&\; \cdots \;\&\; c_n)$ with each c_i either an equality ($p = p'$) or a freshness ($x \# p$). Since such formulas do not involve the relation symbol $r <: A_r$, the satisfaction relation of Definition 2.4 restricts to a relation $V \models \varphi$ between valuations and constraints. A *constraint problem* is a closed constraint formula and it is *satisfiable* if $\emptyset \models \varphi$ holds, where $\emptyset$ denotes the valuation with empty domain.

That satisfaction of α-tree constraint problems is decidable and in NP can be deduced from results about nominal unification [28]: see [5, Theorem 7.1.2]. One can show that it is also NP-hard via the following simple observation, which seems to be new.[1] In stating it we use the abbreviation $\langle x_1, \ldots, x_n \rangle(-)$ to stand for iterated name-abstraction $\langle x_1 \rangle(\cdots \langle x_n \rangle(-) \cdots)$.

Lemma 3.2 (set membership as an α-tree constraint). *Given distinct variables $x, x_1, \ldots, x_k, x', x'_1, \ldots, x'_k \in Var(N)$ (for some name sort N), define $mem(x, x_1, \ldots, x_k) \triangleq \exists x' \exists x'_1 \cdots \exists x'_k\, (x \# x' \;\&\; \langle x_1, \ldots, x_k \rangle x = \langle x'_1, \ldots, x'_k \rangle x')$. Then a valuation V on $\{x, x_1, \ldots, x_k\}$ satisfies $mem(x, x_1, \ldots, x_k)$ iff $V(x)$ is a member of the finite set $\{V(x_i) \mid i = 1..k\}$.* □

We can use this lemma to show NP-hardness by reduction of Graph 3-Colourability. Given a finite graph with vertices $v_1, \ldots, v_n$ (which we can take to be variables of some name sort), edges $e_1, \ldots, e_m$ and source/target functions $s, t : \{e_1, \ldots, e_m\} \rightrightarrows \{v_1, \ldots, v_n\}$, then the formula

[1] Cheney's proof of NP-hardness [4] for his constraint problems is not applicable here, because it relies upon the use of concrete names and name-permutations.

$\exists r, g, b, v_1, \ldots, v_n \left(r \,\#\, g \;\&\; g \,\#\, b \;\&\; b \,\#\, r \;\&\; \&_{i=1}^{n}\, mem(v_i, r, g, b) \;\&\; \&_{j=1}^{m}\, s(e_j) \,\#\, t(e_j) \right)$
is logically equivalent to an α-tree constraint problem which is satisfiable iff the graph's vertices can be coloured with one of three colours (r, g, b) so that no edge connects vertices of the same colour. So altogether we have:

Theorem 3.3. *Satisfiability of α-tree constraint problems is NP-complete.* □

4 αML

We are going to make the α-inductive definitions of Sect. 2 executable by embedding the simple meta-language language of patterns and formulas in which they are expressed within an ML-like functional programming language, called αML. The embedding has two attractive features:

(a) *nominal signatures Σ are subsumed within recursive data type declarations*;

(b) *α-inductive definitions $\mathcal{D}$ become instances of recursively defined functions.*

To achieve point (a) we mimic FreshML [26] and extend ML's type system with types of name N and name-abstraction types `[`N`]`T. However, unlike FreshML and for the reasons given below, we will restrict the use of `[`N`]`T to the case when T is an *equality type* in the sense of Standard ML [20, Sect. 4.4]. To achieve point (b) we note that the meaning $[\![\mathcal{D}]\!]$ of $\mathcal{D}$ is the fixed point of a higher-order function (4). We represent subsets of $\alpha\text{-}Tree_{\Sigma}(A)$ by αML functions of type A `->` `prop`, where `prop` is a new type of "semi-decidable propositions"; and then $\Phi_{\mathcal{D}}$ is represented by a function of type (A `->` `prop`) `->` (A `->` `prop`). In order to be able to write this function, αML extends the pure functional core of ML with name-binding patterns, with equality and freshness constraints, and with existential quantification over values of equality types. The syntax of αML types and expressions is given in Fig. 2. For simplicity's sake αML has a monomorphic type system with a single, top-level data type declaration of some name sorts (N), of some data sorts (S, including a distinguished one `bool` with constructors `T ()` and `F ()`) whose recursive definitions may only involve equality types (E), and of some general data types (D) whose recursive definitions may involve function types and `prop`. Note that in accord with point (a) above, such a declaration subsumes the notion of nominal signature [28] that we used in Sect. 2; in particular, the signature's nominal arities (1) coincide with equality types.

Turning to αML's operational semantics, the behaviour of the pure functional constructs is completely straightforward and could be formalized in any of the standard ways. We use Felleisen-style evaluation contexts [9], formalized using *frame stacks* [22], because this makes the combination with αML's impure features smoother. These impure features are α-tree equality and freshness constraints, and existentially quantified variables of equality type. We describe their behaviour by combining the use of frame stacks with the techniques of *constraint logic programming* (CLP) [15] applied to the α-tree constraint problems of the previous section. A constraint-based approach gives a clean, abstract presentation that avoids the use of unifying substitutions; this is especially useful here

Equality types $E ::=$

S	(data sort)
$E \ast \cdots \ast E$	(tuples, including nullary case, `unit`)
N	(name sort)
$[N]E$	(name-abstraction)

Types $T ::=$

E	(equality type)
D	(data type)
$T \ast \cdots \ast T$	(tuples)
$T \to T$	(functions)
`prop`	(semi-decidable propositions)

Data type declaration:

$$\begin{array}{l}\texttt{names}\ \ N \cdots \\ \texttt{data} \\ \quad \texttt{bool} = \texttt{T of unit} \mid \texttt{F of unit} \\ \quad\quad S = K_1\ \texttt{of}\ E_1 \mid \cdots \mid K_n\ \texttt{of}\ E_n \\ \quad\quad\quad \vdots \\ \quad\quad D = K'_1\ \texttt{of}\ T_1 \mid \cdots \mid K'_{n'}\ \texttt{of}\ T_{n'} \\ \quad\quad\quad \vdots\end{array}$$

Expressions $e, v ::=$

x, f	(variables)
$\texttt{let}\ x = e\ \texttt{in}\ e$	(sequencing)
$K\,v$	(constructor application)
$\texttt{case}\ v\ \texttt{of}\ K\,x \Rightarrow e \mid \cdots \mid K\,x \Rightarrow e$	(case analysis)
$(v, \ldots, v)$	(tuple)
$e.i$	(projections, $i \in \mathbb{N}$)
$\texttt{fun}\ f\ x = e$	(recursive function)
$v\,v$	(function application) *pure*
$\texttt{<}v\texttt{>}v$	(name-abstraction) *impure*
$\texttt{T}$	(empty constraint)
c	(atomic constraint)
$\exists x\, e$	(existential)

Atomic constraints $c ::=$

$v = v$	(equality)
$v \,\#\, v$	(freshness)

Frame stacks $s ::=$

id	(empty)
$s \circ (x.\, e)$	(non-empty)

Expressions and frame stacks in *A-normal form* are obtained by restricting v to range over

Values $v ::=$

$x, f \mid K\,v \mid (v, \ldots, v) \mid \texttt{fun}\ f\ x = e \mid \texttt{<}v\texttt{>}v \mid \texttt{T}$

• *We only consider well-typed expressions and frame stacks.* We use explicitly typed variables $x \in Var(T)$ as T ranges over types. The typing of the pure functional part of αML is entirely standard; the types of αML's impure features are:

Name-abstraction	$\texttt{<}e\texttt{>}e' : [N]E$ if $e : N$ and $e' : E$
Empty constraint	$\texttt{T} : \texttt{prop}$
Equality constraint	$e = e' : \texttt{prop}$ if $e : E$ and $e' : E$
Freshness constraint	$e \,\#\, e' : \texttt{prop}$ if $e : N$ and $e' : E$
Existential	$\exists x\, e : T$ if $x \in Var(E)$ and $e : T$.

• *We identify αML expressions up to renaming bound variables.* Despite the fact that αML is a meta-language for object-level languages with binding, there is no reason not to adopt the usual conventions (see Remark 2.2(a)) for αML's own variable-binding constructs. In the pure part, variable-binding occurs in the usual way, in `let`, `case` and `fun` expressions and in frame stacks $s \circ (x.\, -)$; and in the impure part, $\exists x(-)$ is a binder. We write $FV(e)$ for the finite set of free variables of an expression e and say e is *closed* if this set is empty. We write $e[e'/x]$ for the capture-avoiding substitution of e' for all free occurrences of x in e (well-defined up to renaming bound variables).

Fig. 2. αML syntax

	Pure transitions $s, e \to s', e'$
(P1)	$s \circ (x.\, e), v \to s, e[v/x]$
(P2)	$s, (\texttt{let}\ x = e\ \texttt{in}\ e') \to s \circ (x.\, e'), e$
(P3)	$s, (v_1, \ldots, v_n).i \to s, v_i$ if $i \in \{1..n\}$
(P4)	$s, (\texttt{case}\ K_i\ v\ \texttt{of}\ K_1\ x_1 \Rightarrow e_1 \mid \cdots \mid K_n\ x_n \Rightarrow e_n) \to s, e_i[v/x_i]$ if $i \in \{1..n\}$
(P5)	$s, v\, v' \to s, e[v/f, v'/x]$ if v is $\texttt{fun}\ f\ x = e$
	Impure transitions $\exists \vec{x}(\overline{c}; s; e) \to \exists \vec{x}'(\overline{c}'; s'; e')$
(I1)	$\exists \vec{x}(\overline{c}; s; e) \to \exists \vec{x}(\overline{c}; s'; e')$ if $s, e \to s', e'$
(I2)	$\exists \vec{x}(\overline{c}; s; x.\, i) \to \exists \vec{x}, x_1, \ldots, x_n(\overline{c}\ \&\ x{=}(x_1, \ldots, x_n); s; x_i)$
(I3)	$\exists \vec{x}(\overline{c}; s; \texttt{case}\ x\ \texttt{of}\ K_1\ x_1 \Rightarrow e_1 \mid \cdots \mid K_n\ x_n \Rightarrow e_n) \to \exists \vec{x}, x_i(\overline{c}\ \&\ x{=}K_i x_i; s; e_i)$ if $i \in \{1..n\}$ and $\emptyset \models \exists \vec{x}, x_i(\overline{c}\ \&\ x{=}K_i x_i)$
(I4)	$\exists \vec{x}(\overline{c}; s; c) \to \exists \vec{x}(\overline{c}\ \&\ c; s; \mathsf{T})$ if $\emptyset \models \exists \vec{x}(\overline{c}\ \&\ c)$
(I5)	$\exists \vec{x}(\overline{c}; s; \exists x\, e) \to \exists \vec{x}, x(\overline{c}; s; e)$ if $\emptyset \models \exists x(\mathsf{T})$

• $e, s, e', s', e_i, \ldots$ range over expressions and frame stacks in A-normal form (Fig. 2).
• Impure transitions are between *configurations* $\exists \vec{x}(\overline{c}; s; e)$ where $\overline{c}$ is a finite conjunction of atomic constraints and $\vec{x}$ is a finite list of distinct variables of equality type containing the free variables of $\overline{c}$, s and e. As for expressions, *we identify configurations up to renaming of* $\exists$*-bound variables*. The initial configuration is $\exists \emptyset(\mathsf{T}; id; e)$.
• In (I2) $x \in Var(E_1 * \cdots * E_n)$ and $x_i \in Var(E_i) - \vec{x}$ for $i = 1..n$.
• In (I3) $x \in Var(S)$, $S = K_1\ \texttt{of}\ E_1 \mid \cdots \mid K_n\ \texttt{of}\ E_n$ and $x_i \in Var(E_i) - \vec{x}$ for $i = 1..n$.
• In (I5) $x \notin \vec{x}$. If $x \in Var(E)$ say, then constraint $\exists x(\mathsf{T})$ is satisfiable iff E is non-empty, in the sense that there is an α-tree of arity E. We allow empty data sorts, e.g. that given by the declaration $\texttt{es} = K\ \texttt{of}\ \texttt{es}$.

Fig. 3. αML operational semantics

because the "possibly-capturing" nature of substitution (cf. Remark 2.2(d)) complicates unification—see for example the use of terms involving explicit name-permutations in nominal [28] and equivariant [6] unification algorithms. αML's operational semantics is specified in Fig. 3. To simplify the presentation we have restricted to the *A-normal forms* [10] from Fig. 2; transitions for general expressions can be derived by reducing them to A-normal form. The αML transition relation is non-deterministic, because of the "narrowing" that occurs when evaluating a `case`-expression whose subject is an existentially quantified variable (transition (I3) in Fig. 3). There is a considerable literature about this specific source of non-determinism, centred around the semantics of the functional logic programming language Curry; see [14] for a survey. Since αML features non-trivial computational effects and we do not wish to impose a monadic programming style, we prefer a strict evaluation strategy, rather than the call-by-need strategy that is more common in the functional logic programming literature; and for simplicity's sake we wish to avoid residuation and concurrent execution [14, Sect. 2.4]. So we use a simple-minded design where the "rigid/flexible" behaviour of `case`-analysis is part of the dynamics (pure transition (P4) versus impure transition (I3)), rather than user-specified.

The following theorem shows that we do achieve the design goal of embedding within αML the usual operational behaviour of pure functional programming

with recursive data types and call-by-value higher-order functions. It depends on a notion of configurations being well-typed, $\exists\vec{x}(\overline{c}; s; e) : T$, whose straightforward definition we omit here.

Theorem 4.1 (embedded pure functional language). *An αML expression or frame stack is* pure *if it does not contain sub-expressions of the form* $\texttt{<}e\texttt{>}e'$, T, $e = e'$, $e \,\#\, e'$, *or* $\exists x\, e$. *Suppose s and e are pure and that $\exists\emptyset(\mathsf{T}; s; e) : T$ holds for some type T. Then $\exists\emptyset(\mathsf{T}; s; e) \to \exists\vec{x}(\overline{c}; s'; e')$ holds iff $\vec{x} = \emptyset$, $\overline{c} = \mathsf{T}$, s' and e' are pure, and there is a pure transition $(s; e) \to (s'; e')$.* □

αML restricts the name-abstraction type-former $\texttt{[}N\texttt{]}(-)$ to apply only to equality types E (that is, to nominal arities) rather than to general types T, for which equality constraints are in general uncomputable. This allows expressions of name-abstraction type to be deconstructed by unification with name-abstraction patterns $\texttt{<}x\texttt{>}p$ in the presence of freshness constraints on x, rather than using the "generative name unbinding" [25] mechanism of FreshML, which is based on a supply of dynamically allocated fresh names ("gensym"). Here is an example.

Example 4.2. If the nominal signature for λ-terms mentioned in Sect. 2 is part of the data type declaration, then the λ-term substitution operation $(t, t', x') \mapsto t[t'/x']$ can be encoded in αML by the following function of type `tm*tm*vr -> tm`

$$\begin{array}{l}\texttt{fun}\ \mathit{subst}(t, t', x') = \texttt{case}\ t\ \texttt{of}\\ \quad \texttt{V}\, x \Rightarrow \texttt{if}\ x = x'\ \texttt{then}\ t'\ \texttt{else}\ t\\ \texttt{|}\ \texttt{A}\, x \Rightarrow \texttt{A}(\mathit{subst}(x.1, t', x'), \mathit{subst}(x.2, t', x'))\\ \texttt{|}\ \texttt{L}\, x \Rightarrow \exists x_1\, \exists t_1\ \underline{x_1 \,\#\, (t', x')}\ \texttt{\&}\ \texttt{<}x_1\texttt{>}t_1 = x\ \texttt{\&}\ \texttt{L}(\texttt{<}x_1\texttt{>}\mathit{subst}(t_1, t', x'))\end{array} \tag{5}$$

where we have used some syntactic sugar for tuple-pattern matching, together with the following abbreviations:

$$e_1\ \texttt{\&}\ e_2 \ \triangleq\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \tag{6}$$

$$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \ \triangleq\ \texttt{case}\ e\ \texttt{of}\ \texttt{T}\, x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{F}\, x_2 \Rightarrow e_2 \tag{7}$$

(where $x \notin FV(e_2)$ in (6) and $x_i \notin FV(e_i)$ in (7)). The underlined freshness constraint in (5) enforces the usual "capture-avoiding" property when substituting under a λ-binder (cf. [7, Example 2.3]).

Remark 4.3 (dynamically allocated names). We can add dynamically allocated names to αML without breaking the "names as meta-variables" aspect of its design (Remark 2.2(b)): extend its syntax with expressions $\texttt{fresh}_N$ of type N (for each name sort N) and its operational semantics with the impure transition:

$$\text{(I6)} \qquad \exists\vec{x}(\overline{c}; s; \texttt{fresh}_N) \to \exists\vec{x}, x(\overline{c}\ \texttt{\&}\ x \,\#\, \vec{x}; s; x)$$

where $x \in Var(N)$ is not in $\vec{x}$ and $x \,\#\, \vec{x}$ is the constraint $x \,\#\, x_1\ \texttt{\&}\ \cdots\ \texttt{\&}\ x \,\#\, x_n$ when $\vec{x} = x_1, \ldots, x_n$. Using this we can define a uniform operation of *generative name-unbinding*

$$\texttt{unbind}\, e\ \texttt{as}\ \texttt{<}x\texttt{>}x'\ \texttt{in}\ e' \ \triangleq\ \texttt{let}\ x = \texttt{fresh}_N\ \texttt{in}\ \exists x'\, \texttt{<}x\texttt{>}x' = e\ \texttt{\&}\ e' \tag{8}$$

where $e : \texttt{[}N\texttt{]}E$ and $x \in Var(N), x' \in Var(E)$ are distinct variables not occurring in e. Then, for example, we can replace the last branch of the `case` expression in (5) with $\texttt{L}\,x \Rightarrow \texttt{unbind}\,x\,\texttt{as <}x_1\texttt{>}t_1\,\texttt{in L(<}x_1\texttt{>}subst(t_1, t', x')\texttt{)}$. Rule (I6) and definition (8) together give a version of generative unbinding that is like the one used by MLSOS [16]. It is operationally different from FreshML's version [26], which pushes a swap of x with a fresh name into the body e'. Are these two forms of generative unbinding behaviourally equivalent? To determine this requires developing the properties of *contextual equivalence* for αML, which we defer to a future paper. Is $\texttt{fresh}_N$ definable up to contextual equivalence in terms of the language presented in Figs 2 and 3? It seems unlikely, but we do not have a proof.

5 α-Inductive Definitions as αML Recursive Functions

We remarked in the previous section that nominal arities are the same thing as αML equality types. Regarding the relation symbol $r <: A_r$ of nominal arity A_r as a variable of type $A_r \texttt{->} \texttt{prop}$, we identify α-inductive definitions in standard form $\mathcal{D}$ (Definition 2.5) with certain αML recursive function values $v_{\mathcal{D}}$ of type $A_r \texttt{->} \texttt{prop}$:

$$v_{\mathcal{D}} \triangleq (\texttt{fun}\, r\, x = \varphi) \quad \text{where } \mathcal{D} \text{ is } \frac{\varphi}{r(x)}. \tag{9}$$

For this to make sense we have to embed formulas $\varphi \in Form_{\Sigma}$ over a nominal signature Σ (Definition 2.4) as αML expressions of type `prop`. Clearly the patterns p of each arity A (Definition 2.1) coincide with values v (Fig. 2) of equality type A. So αML syntax has all the necessary constituents for expressing formulas except possibly for conjunction and disjunction. We define conjunction as in (6) and express disjunction using a flexible `case`-expression (cf. [27, Sect. 3.1]):

$$\begin{aligned} e_1 \texttt{ v } e_2 \triangleq\ & \exists x\,(\texttt{case}\,x\,\texttt{of T}\,x_1 \Rightarrow e_1 \texttt{ | F}\,x_2 \Rightarrow e_2) \\ & \text{(where } x, x_1, x_2 \notin FV(e_1, e_2)\text{).} \end{aligned} \tag{10}$$

So given an α-inductive definition $\mathcal{D}$ in standard form with associated αML function $v_{\mathcal{D}} : A_r \texttt{->} \texttt{prop}$ as in (9), for each formula $\varphi' \in Form_{\Sigma}$ we get an αML expression $\varphi'[v_{\mathcal{D}}/r]$ of type `prop`. The following theorem characterizes satisfaction of φ' in terms of the operational behaviour of this expression. It uses the *solution set* of φ' (with respect to a set of variables $\vec{x}$ containing those free in φ'): this is defined to be the set $solns(\varphi')$ of constraints $\exists\vec{x}'(\overline{c})$ such that $\exists\vec{x}(\texttt{T}; id; \varphi'[v_{\mathcal{D}}/r]) \rightarrow\cdots\rightarrow \exists\vec{x}, \vec{x}'(\overline{c}; id; \texttt{T})$ and $\emptyset \models \exists\vec{x}, \vec{x}'(\overline{c})$.

Theorem 5.1. *For any formula φ' and valuation V we have:*

Soundness: *if $\exists\vec{x}'(\overline{c}) \in solns(\varphi')$ and $V \models \exists\vec{x}'(\overline{c})$ then $([\![\mathcal{D}]\!], V) \models \varphi'$.*

Completeness: *if $([\![\mathcal{D}]\!], V) \models \varphi'$ then there is $\exists\vec{x}'(\overline{c}) \in solns(\varphi')$ with $V \models \exists\vec{x}'(\overline{c})$.*

Proof. The theorem can be deduced using standard techniques from constraint logic programming (CLP) [15, Sect. 4.4 and 4.5]. This is because one can prove that αML's operational semantics agrees with the semantics of CLP goal states $\langle \varphi, \varphi_1, \ldots, \varphi_n \,|\, \overline{c} \rangle$ if the latter are encoded as αML configurations of the form $\exists \vec{x}(\overline{c}; s_{\varphi_n,\ldots,\varphi_1}; \varphi)$, where the frame stack $s_{\varphi_n,\ldots,\varphi_1}$ is defined by: $s_\emptyset \triangleq id$ and $s_{\vec{\varphi},\varphi} \triangleq s_{\vec{\varphi}} \circ (x.\,\varphi)$ (where $x \notin FV(\varphi)$). □

Definition 5.2 (success). We say that a configuration $\exists \vec{x}(\overline{c}; s; e) : \texttt{prop}$ *may succeed* and write $\exists \vec{x}(\overline{c}; s; e)\downarrow$ if there is a finite sequence of transitions from $\exists \vec{x}(\overline{c}; s; e)$ to a configuration of the form $\exists \vec{x}, \vec{x}'(\overline{c}'; id; \mathsf{T})$, for some $\vec{x}'$ and $\overline{c}'$ with $\emptyset \models \exists \vec{x}, \vec{x}'(\overline{c}')$.

We can use Theorem 5.1 to deduce that the operational semantics of $v_{\mathcal{D}} : A_r \texttt{->} \texttt{prop}$ in αML detects, through the above notion of success, all and only the α-trees $t \in \alpha\text{-}Tree(A_r)$ lying in the inductively defined subset $[\![\mathcal{D}]\!]$ (Definition 2.5). To do so, we first have to discuss how αML represents α-trees, since they involve concrete names $n \in Name(N)$ whereas αML follows the common practice (Remark 2.2(b)) of only using variables of name sort, $x \in Var(N)$. What matters about names when they are used to describe binding structure is not their particular identity, but rather the *distinctions* between them—and those can be expressed using constraints asserting that all the variables in a list $\vec{x} = x_1, \ldots, x_k$ are distinct:

$$\#_{\vec{x}} \triangleq \mathop{\&}_{1 \le i < j \le k} x_i \,\#\, x_j.$$

A valuation V with domain $\vec{x}$ satisfies $\#_{\vec{x}}$ iff $V(x_1), \ldots, V(x_k)$ are (α-equivalence classes of) mutually distinct names. We can represent a particular α-tree in αML by a pattern in the presence of such a constraint: if $t \in \alpha\text{-}Tree(A_r)$ is the α-equivalence class of a tree involving k distinct names (bound or free), we can find a pattern $p : A_r$ with k variables $\vec{x}$ of name sort and a valuation V with $t = [\![p]\!]_V$ and $V \models \#_{\vec{x}}$. Then taking φ' to be $\#_{\vec{x}} \,\&\, r(p)$ in Theorem 5.1 we get:

Corollary 5.3. *Let $\mathcal{D}$ be an α-inductive definition in standard form with associated αML function $v_{\mathcal{D}} : A_r \texttt{->} \texttt{prop}$. If $t \in \alpha\text{-}Tree(A_r)$ is represented as above by a pattern $p : A_r$ and a valuation V (with $dom(V) = \vec{x}$, the variables of p), then $t \in [\![\mathcal{D}]\!]$ iff $\exists \vec{x}(\#_{\vec{x}}; id; v_{\mathcal{D}}\, p)\downarrow$.* □

6 Related and Future Work

A popular approach to executable operational semantics is to use *higher-order logic programming*, where binders in inductive definitions are represented via higher-order abstract syntax (HOAS): see Miller [19] for an overview. We think it is both useful and interesting to study executable operational semantics also using *functional logic programming* [14]. It has proved harder to integrate HOAS representations with functional programming: see [18] for a recent view on this. In any case, in the Introduction we advocated leaving the HOAS mainstream and pursuing a nominal approach, for reasons to do with name-aliasing. As far as we know the

first such approach was Cheney and Urban's first-order logic programming language αProlog [7]. Our first attempt to combine αProlog's computational mechanism (resolution based on nominal unification [28]) with higher-order typed functional programming was influenced by the work on FreshML [26] and produced MLSOS [16]. Byrd and Friedman's αKanren [2] combines it with the untyped functional language Scheme. αProlog, MLSOS and αKanren allow the use of constants to name object-level bound entities. We argued in Remark 2.2(b) that such concrete names are never used in practice when specifying inductively defined relations. Moreover their use naturally leads to "equivariance" (that is, invariance under permutations of concrete names) becoming an explicit part of the meta-language's operational semantics [6], rather than just a useful meta-theoretic property of the semantics. By contrast, αML only uses *meta-variables* rather than constants to name object-level bound entities, plus freshness constraints on meta-variables when distinctions between names are needed. As well as being closer to informal practice, this approach leads both to a pleasingly simple design for αML's operational semantics (Fig. 3) and a correctness result (Corollary 5.3) that was lacking for MLSOS. Our design also avoids the use of explicit name-swapping; although this is a characteristic feature of nominal logic [23], nominal unification and αProlog, it is not needed from the point of view of a user specifying operational semantics in an executable meta-language.

The presence of unifiable meta-variables in binding position in αML patterns does mean that, as for Cheney's equivariant unification [4], our constraint satisfaction problem is NP-complete (Theorem 3.3). There are at least two different approaches to obtaining a practically useful implementation of αML that should be investigated. One approach is to identify restrictions on α-inductive definitions that do not limit their applicability for specifying operational semantics too much, but for which the associated α-tree constraint problems are in P rather than NP; cf. Cheney and Urban [7, Sect. 5.3]. Since degrees of "applicability for specifying operational semantics" are hard to pin down, perhaps a more attractive alternative is to stick with the general and conceptually simple form of α-inductive definitions used in this paper, but investigate transformations on α-tree constraint problems that allow the highly developed technology of SAT-solvers to be applied.

References

[1] Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. North-Holland, Amsterdam (1984) (revised edition)
[2] Byrd, W.E., Friedman, D.P.: alphaKanren: A fresh name in nominal logic programming. In: Proc. 2007 Workshop on Scheme and Functional Programming, number DIUL-RT-0701 in Université Laval Technical Reports, pp. 79–90 (2007)
[3] Calvès, C., Fernández, M.: Nominal matching and alpha-equivalence. In: Hodges, W., de Queiroz, R. (eds.) Logic, Language, Information and Computation. LNCS, vol. 5110, pp. 111–122. Springer, Heidelberg (2008)
[4] Cheney, J.: The complexity of equivariant unification. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 332–344. Springer, Heidelberg (2004)
[5] Cheney, J.: Nominal Logic Programming. PhD thesis, Cornell Univ. (2004)

[6] Cheney, J.: Equivariant unification. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 74–89. Springer, Heidelberg (2005)
[7] Cheney, J., Urban, C.: Nominal logic programming. Trans. Prog. Lang. and Systems 30(5), 1–47 (2008)
[8] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indag. Math. 34, 381–392 (1972)
[9] Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoret. Comput. Science 103, 235–271 (1992)
[10] Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. 28, 237–247 (1993)
[11] Gabbay, M.J.: Fresh logic: Proof-theory and semantics for FM and nominal techniques. J. Appl. Logic 5, 356–387 (2007)
[12] Gordon, A.D.: Operational equivalences for untyped and polymorphic object calculi. In: Gordon and Pitts [13], pp. 9–54
[13] Gordon, A.D., Pitts, A.M. (eds.): Higher Order Operational Techniques in Semantics. Cambridge University Press, Cambridge (1998)
[14] Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
[15] Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: Semantics of constraint logic programming. J. Logic Programming 37, 1–46 (1998)
[16] Lakin, M.R., Pitts, A.M.: A metalanguage for structural operational semantics. In: Trends in Functional Programming, vol. 8, pp. 19–35. Intellect (2008)
[17] Lassen, S.B.: Relational reasoning about contexts. In: Gordon and Pitts [13], pp. 91–135
[18] Licata, D.R., Zeilberger, N., Harper, R.: Focusing on binding and computation. In: LICS 2008 Proceedings, pp. 241–252. IEEE Computer Society, Los Alamitos (2008)
[19] Miller, D.A.: Abstract syntax for variable binders: An overview. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 239–253. Springer, Heidelberg (2000)
[20] Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
[21] Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: PLDI 1988 Proceedings, pp. 199–208. ACM Press, New York (1988)
[22] Pitts, A.M.: Operational semantics and program equivalence. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 378–412. Springer, Heidelberg (2002)
[23] Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. and Comput. 186, 165–193 (2003)
[24] Pitts, A.M.: Alpha-structural recursion and induction. J. ACM 53, 459–506 (2006)
[25] Pitts, A.M., Shinwell, M.R.: Generative unbinding of names. Logical Methods in Comput. Science 4, 1–33 (2008)
[26] Shinwell, M.R., Pitts, A.M., Gabbay, M.J.: FreshML: Programming with binders made simple. In: ICFP 2003 Proceedings, pp. 263–274. ACM Press, New York (2003)
[27] Tolmach, A., Antoy, S.: A monadic semantics for core Curry. In: WFLP 2003 Proceedings. Electr. Notes in Theoret. Comp. Science, vol. 86(3), pp. 16–34. Elsevier, Amsterdam (2003)
[28] Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoret. Comp. Science 323, 473–497 (2004)