Chapter 1

A metalanguage for structural operational semantics

Matthew Lakin[1], Andrew Pitts[1]

Abstract: This paper introduces MLSOS, a functional metalanguage for declaring and animating definitions of structural operational semantics. The language provides a general mechanism for resolution-based search that respects the α-equivalence of object-language binding structures, based on nominal unification. It combines that with a FreshML-style generative treatment of bound names. We claim that MLSOS allows animation of operational semantics definitions to be prototyped in a natural way, starting from semi-formal specifications. We outline the main design choices behind the language and illustrate its use.

1.1 INTRODUCTION

There is currently a great deal of interest in (partially) automating various tasks in the field of programming language metatheory. While there are many aspects to this research effort (see [1] for a survey), the work reported here focuses on animating definitions of type systems and operational semantics. This typically involves generating a reference or prototype implementation from a high-level description of the desired semantics. General-purpose functional programming languages such as Objective Caml and Haskell are usually the first choice for such an implementation. However, such languages do not provide built-in support for representing object-language binders up to α-equivalence or for "proof-search" style computations of the validity of some judgement (such as the type inference problem) given a rule-based inductive definition of that judgment. Hence, the programmer must "reinvent the wheel" and re-implement this basic functionality every time they want to implement a language. This takes up time and increases

[1]University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK. Email addresses: Matthew.Lakin@cl.cam.ac.uk, Andrew.Pitts@cl.cam.ac.uk.

the potential for bugs to appear. Furthermore, in the early stages of language design, the ability to play with a toy implementation of the language can be invaluable. It can provide useful feedback on test cases and guide the evolution of the language, with only a fraction of the effort required for a full formal verification of the language metatheory.

Our long-term aim is to automatically generate correct, efficient, executable code from a high-level specification of the intended behaviour, such as the inference rules for inductively defined relations. The first thing that we need, however, is a programmable metalanguage that allows the user to succinctly specify that behaviour. An important issue that must be faced in programming language support for computing instances of inductively defined relations is how to implement bound names in the object-language in a way that deals with issues of α-equivalence automatically. To that end we make use of *nominal unification* [23]—a generalisation of first-order unification that solves equations modulo α-equivalence by taking into account *freshness* of names for object-language terms.

We propose an eager functional programming language for resolution-based computations on abstract syntax up to α-equivalence of object-language bound names, based on a mild generalisation of nominal unification. We call this language MLSOS (**M**eta**L**anguage for **S**tructural **O**perational **S**emantics). One of the main contributions of the design is to show how to combine, in a well-behaved way, nominal unification's logical treatment of freshness with a dynamic approach to freshness derived from the FreshML programming language [19]. The *"Barendregt variable convention"* [3] expresses (informally) that bound names should be mutually distinct and distinct from other names appearing in the mathematical context. Hence, when it attempts to unify against a pattern containing binders, MLSOS automatically enforces the user's expectation that the Barendregt variable convention holds by replacing bound names with freshly generated names as the pattern is analysed. This "generative unbinding" of binders is the characteristic feature of FreshML. On the other hand, when it encounters names in a pattern that are not bound, MLSOS uses unification variables, since it would be premature to commit to any particular name in this case. Hence our motto is "implement bound names generatively, and free names with unification variables". We claim that this leads to a style of programming which, compared to the logic programming language αProlog [5] that also makes use of nominal unification, relieves the user of the need to specify much information about the freshness of names in operational semantics specifications. (See §1.3.1 and §1.5 for evidence of this.)

The rest of this paper is organised as follows. §1.2 provides a brief overview of nominal unification. §1.3 gives an extended informal example of the use of MLSOS. §1.4 goes into the formalities, defining the grammar and operational semantics of the core MLSOS language. §1.5 discusses related work; we describe future work and conclude in §1.6.

1.2 BACKGROUND: NOMINAL UNIFICATION

Our method of specifying binding is the *nominal signature* of [23], which consists of a finite set of *atom sorts* α (types of object-language names) and a finite set of *data sorts* δ (types of α-equivalence classes of object-language terms). We build up a set of *arities* σ by the grammar

$$\sigma ::= \alpha \mid \delta \mid 1 \mid \sigma_1 * \cdots * \sigma_n \mid \langle\langle\alpha\rangle\rangle\sigma.$$

The arity $\sigma_1 * \cdots * \sigma_n$ is for n-tuples of terms, $(t_1,\ldots,t_n)$. The arity $\langle\langle\alpha\rangle\rangle\sigma$ is where binding is specified—it is inhabited by terms $\langle\langle a\rangle\rangle t$ representing the object-language binding of a *single* name a of atom sort α in a term t of arity σ. Given this grammar of arities, the specification of a nominal signature is completed by giving a finite set of typed *constructors*, $\mathsf{K} : \sigma \to \delta$, which allows us to construct terms $\mathsf{K}t$ of data sort δ from terms t of arity σ. By way of an example, here is a nominal signature for untyped λ-terms, as it is declared in MLSOS:

```
nametype var ;;
datatype lam = Var of var
             | Lam of ⟨⟨var⟩⟩lam
             | App of lam * lam ;;
```

Atom sorts (such as var) are declared using the nametype keyword; whereas data sorts (such as lam) and constructors (such as Var, Lam and App) are declared with an ML-like datatype declaration.

The *nominal terms* of the various arities over a given nominal signature are built up using the term-forming operations mentioned above, starting from a unit value `()` of arity 1, from countably many *atoms* a of each atom sort, and from countably many *suspensions* πX for each arity; the latter consist of a unification variable X and a finite permutation π of atoms waiting to be applied to whatever will be substituted for X. (See §1.4.1 and [23] for the need for such suspended permutations.)

Nominal unification is an algorithm for unifying nominal terms up to α-equivalence. As shown in [23], to make sense of α-equivalence for nominal terms, one needs to consider *freshness conditions* of the form $a \# X$ (read "a fresh for X") whose intended meaning is that a should not occur free in any term substituted for X. Thus given a unification problem $t \mathrel{=:=} t'$ (where t and t' are nominal terms of the same arity over a given nominal signature), the nominal unification algorithm computes both a solving substitution of nominal terms for unification variables and a set of freshness conditions under which the required α-equivalence is valid (or fails finitely if no such solution exists). For example, it solves the problem $\langle\langle a\rangle\rangle X \mathrel{=:=} \langle\langle b\rangle\rangle Y$ with the substitution $[Y \mapsto (a\,b)\,X]$ and the freshness condition $b \# X$. This means that the unification variable Y should be instantiated to the result of *swapping* all occurrences of a and b throughout the term eventually substituted for X. The freshness constraint that b may not occur free in X prevents name capture. We refer the reader to [23] for more details.

$$(\mathsf{beta}_1)\ \frac{}{\mathsf{beta}\ (t,\ t)} \qquad (\mathsf{beta}_2)\ \frac{\mathsf{beta}\ (t_1,\ t_1') \qquad \mathsf{beta}\ (t_2,\ t_2')}{\mathsf{beta}\ (t_1\ t_2,\ t_1'\ t_2')}$$

$$(\mathsf{beta}_3)\ \frac{\mathsf{beta}\ (t,\ t')}{\mathsf{beta}\ (\lambda x.t,\ \lambda x.t')} \qquad (\mathsf{beta}_4)\ \frac{\mathsf{beta}\ (t_1,\ t_1') \qquad \mathsf{beta}\ (t_2,\ t_2')}{\mathsf{beta}\ ((\lambda x.t_1)\ t_2,\ t_1'[t_2'/x])}$$

FIGURE 1.1. Parallel Reduction Relation

```
1   let rec sub x t t' = narrow t' as
2       Var y → ((x =:= y); t) or ((x =/= y); t')
3     | Lam ⟨⟨a⟩⟩t'' → Lam ⟨⟨a⟩⟩(sub x t t'')
4     | App (t1, t2) → App ((sub x t t1), (sub x t t2)) ;;

5   let rec beta (t1, t2) = narrow (t1, t2) as
6       (t, t) → yes
7     | (Lam ⟨⟨a⟩⟩t1, Lam ⟨⟨a⟩⟩t1') → beta (t1, t1')
8     | (App (t1, t2), App (t1', t2')) → beta (t1, t1'); beta (t2, t2')
9     | (App ((Lam ⟨⟨a⟩⟩t1), t2), t') → some t1', t2' : lam in
10          beta (t1, t1'); beta (t2, t2'); ((sub a t2' t1') =:= t') ;;
```

FIGURE 1.2. MLSOS Parallel Reduction Program

Nominal unification has polynomial complexity [4] and produces most general unifiers [23]. That the latter exist depends upon the fact that nominal terms only allow abstractions over concrete atoms, $\langle\langle a\rangle\rangle t$, but not abstractions over unification variables, $\langle\langle X\rangle\rangle t$. As we shall see, this restriction greatly simplifies constraint solving. Without it, we would need to use a more powerful but **NP**-complete *equivariant* unification algorithm due to Cheney [8, 6].

1.3 PROGRAMMING EXAMPLE: PARALLEL REDUCTION

Figure 1.1 gives inference rules defining the beta relation of "parallel reduction" (that is, reducing several β-redexes within a λ-term in one go). This relation is used in the Tait/Martin-Löf proof of the Church-Rosser property of β-reduction in the untyped λ-calculus [3, Definition 3.2.3].

Using the nominal signature from the previous section, Figure 1.2 gives ML-SOS functions sub : var → lam → lam → lam and beta : lam ∗ lam → ans, where ans is the MLSOS built-in type for proof-search computations—it is a copy of the unit type whose only value, yes, indicates success. The first implements capture-avoiding substitution for λ-terms and the second implements the parallel reduction

relation. This section provides a brief explanation of how the code from Figure 1.2 would evaluate. This code uses various syntactic sugars which are implemented by translation into a small core language described later in this paper (§1.4).

1.3.1 Capture-avoiding substitution

Evaluating sub $x\ t\ t'$ (Figure 1.2, lines 1–4) computes the *capture avoiding* substitution of (the α-equivalence class of) t for all free occurrences of the name x in (the α-equivalence class of) t'.

The narrow syntax is syntactic sugar for a non-deterministic branch, which here generates three branches of computation. The patterns in the individual clauses are expanded out into expressions which generate fresh atoms and unification variables as required: as discussed in the Introduction, we use fresh atoms to implement bound names and unification variables to stand for all other unknowns.

Each branch generates a constraint that t' should unify with the patterns on the left-hand sides of lines 2–4 to decide whether to proceed, instantiating unification variables in t' if necessary. The variable clause (line 2) also has an explicit branch (using the or keyword) which is used to encode the standard name equality test. We use a definable syntactic sugar $(e;\ e')$ for sequencing, to simulate a conjunction.

The nominal unification algorithm [23] is used to decide satisfiability of equality constraints, along with some extra rules to cope with name inequality constraints. These are largely straightforward, except for the case when two unknown names are constrained to be distinct (discussed in §1.4).

The clause for λ-abstractions (Figure 1.2, line 3) highlights the different behaviour of value identifiers and atom identifiers in patterns in MLSOS: in line 2, y becomes a new unification variable, whereas in line 3, a is replaced with a *fresh* atom, because it appears in binding position. This means that the informal "Barendregt variable convention" [3] is handled implicitly at the language level, and hence we do not need to decorate the code with assertions that a must not appear free in t (or be equal to x).

1.3.2 Parallel reduction

The beta function declared in lines 5–10 of Figure 1.2 implements the relation inductively defined in Figure 1.1. Line 6 expresses the base case (beta_1) of the definition, line 7 the rule (beta_3), line 8 the rule (beta_2) and lines 9–10 the rule (beta_4). Again, the atom identifier a in the pattern $(\mathsf{App}\ ((\mathsf{Lam}\ \langle\langle a\rangle\rangle t_1),\ t_2),\ t')$ in line 9 refers to a *fresh* atom—this, along with the behaviour of the sub function, ensures that the substitution involved in the MLSOS version of rule (beta_4) is not capturing. Note the use of the some syntax in line 9 to generate unification variables standing for unknown intermediate terms, and the use of sequencing to model a conjunction.

? fresh a : var ;;
− : var = var_0

? fresh b : var ;;
− : var = var_1

? let t_1 = App ((Lam $\langle\langle a \rangle\rangle$Lam $\langle\langle b \rangle\rangle$Var a), Var b) ;;
− : lam = App (Lam $\langle\langle \mathsf{var}_0 \rangle\rangle$Lam $\langle\langle \mathsf{var}_1 \rangle\rangle$Var var_0, Var var_1)

? let t_2 = Lam $\langle\langle a \rangle\rangle$Var b ;;
− : lam = Lam $\langle\langle \mathsf{var}_0 \rangle\rangle$Var var_1

? beta (t_1, t_2) ;;
− : ans = yes

? let t = App ((Lam $\langle\langle a \rangle\rangle$Var a), Var b) ;;
− : lam = App (Lam $\langle\langle \mathsf{var}_0 \rangle\rangle$Var var_0, Var var_1) ;;

? some x : lam ;;
− : lam = unknown

? beta (t, x) ;;
− : ans = yes [x = Var var_1]
− : ans = yes [x = App (Lam $\langle\langle \mathsf{var}_2 \rangle\rangle$Var var_2, Var var_1)]
− : ...

FIGURE 1.3. Command-Line Example

1.3.3 Command-line example

Figure 1.3 illustrates a typical interaction with the MLSOS interpreter. We assume here that the nominal signature for λ-terms from §1.1 and the functions from Figure 1.2 have already been declared earlier in the session.

The first two interactions generate (distinct) fresh atoms a and b, to stand for variables in the λ-calculus. In all cases, the responses from the interpreter include tags such as var_0, to allow the user to identify the atoms generated internally during expression evaluation. We then construct two λ-terms t_1 and t_2, corresponding to $(\lambda a.\lambda b.a)\ b$ and $\lambda a.b$ respectively, and ask the system whether beta $(((\lambda a.\lambda b.a)\ b),\ \lambda a.b)$ is derivable using the rules from Figure 1.2. This is clearly the case (using the final rule (beta_4) of the definition of beta), and indeed the interpreter responds `yes`, as we would expect.

We then define a term t corresponding to $(\lambda a.a)\ b$, and generate a new unification variable (which we bind to the value identifier x). The final command instructs the interpreter to find all instantiations of x for which beta $((\lambda a.a)\ b,\ x)$ holds. This produces numerous answers: the standard β-reduction to Var b, the trivial case arising from the clause allowing any term to β-reduce to itself, and duplicate results caused by redundancies in the rules defining the beta relation. The var_2 tags appearing in the results of this computation are due to the dynamic generation of atoms during proof-search.

These examples give a flavour of how MLSOS might be used. In particular, the final example illustrates the use of unification variables to search for terms (up to α-equivalence) for which some judgement can be derived.

1.4 MLSOS CORE LANGUAGE AND OPERATIONAL SEMANTICS

This section defines the MLSOS "core" language and its operational semantics. We restrict ourselves to the core language since it is small and elegant and the various syntactic sugars employed in Figure 1.2 may be defined in it.

We fix countably infinite sets $\mathbb{V}$ of value identifiers (ranged over by x, y etc.), $\mathbb{A}$ of atoms (ranged over by a, b etc.) and $\mathbb{U}$ of unification variables (ranged over by X). These stand for metalanguage values, object-language names and unknown object-language terms respectively. The grammar of MLSOS types τ is

$$\tau ::= \sigma \mid \mathsf{ans} \mid \tau \to \tau$$

where σ ranges over nominal arities as defined in §1.1. The type ans is a version of ML's unit type that we use for proof-search computations when only the success of the search rather than some final value is important. For example, semi-decidable relations of arity σ are typically implemented in MLSOS as functions of type $\sigma \to \mathsf{ans}$.

1.4.1 Permutations

Nominal logic [14] is based on the fundamental operation of *permuting* atoms. One only needs to consider finite permutations, that is, bijections on the set of atoms that only move finitely many atoms. We use a concrete representation for such permutations as finite lists of atom-swappings, which are written $(a\,a')$. Such a finite list represents the composition of the individual swappings, and it can be shown that every finite permutation can be represented in this way. We write Perm for the set of all *well-formed* atom-permutations, ranged over by π. A permutation is well-formed if, for every swapping $(a\,a')$ in π, the atoms a and a' are of the same atom-sort.

Unknown object-language terms are represented by *suspensions*, written πX. These represent a permutation π waiting to be applied to all atoms appearing in whatever term gets grafted in place of the unification variable X. These suspended permutations are used to ensure that α-conversion behaves correctly in the presence of unification variables: see [23].

1.4.2 Values and expressions

Figure 1.4 gives grammars for the values and expressions of the MLSOS core language. The values are as one would expect from a functional language, with the addition of suspensions (to stand for unknown object-language terms), atoms and atom abstractions, and the yes value (which reports success in a proof-search computation).

v	$::=$	x	value identifier,
	$\mid$	πX	suspension,
	$\mid$	$()$	unit,
	$\mid$	$(v_1, \ldots, v_n)$	n-tuple,
	$\mid$	$\mathsf{fun}\, f(x : \tau) : \tau' = e$	recursive function,
	$\mid$	yes	success,
	$\mid$	$\mathsf{K}\, v$	data construction,
	$\mid$	a	atom,
	$\mid$	$\langle\langle a \rangle\rangle v$	atom abstraction.
e	$::=$	v	value,
	$\mid$	$\mathsf{let}\, x = e\, \mathsf{in}\, e'$	let-binding,
	$\mid$	$v\, v'$	function application,
	$\mid$	$\mathsf{fresh}\, a : \alpha\, \mathsf{in}\, e$	fresh atom,
	$\mid$	$\mathsf{some}\, x : \sigma\, \mathsf{in}\, e$	new unification variable,
	$\mid$	c	constraint,
	$\mid$	$e\, \mathsf{or}\, e'$	binary branch.

FIGURE 1.4. MLSOS Values And Expressions

The expression grammar is structured so that MLSOS core programs must be in A-normal form [10]—that is, evaluation is driven by let-bindings and all results of intermediate computations must be named. Programs written in a more liberal language can be translated into this format by the insertion of additional let-bindings. This restriction further simplifies the presentation of the operational semantics (see §1.4.4). The expression grammar includes constructs for generating fresh atoms ($\mathsf{fresh}\, a : \alpha\, \mathsf{in}\, e$) and new unification variables ($\mathsf{some}\, x : \sigma\, \mathsf{in}\, e$), branching ($e\, \mathsf{or}\, e'$) and testing constraints for satisfiability (c). Note that the programmer has no direct access to unification variables and atoms: like the treatment of mutable reference cells in traditional functional programming languages, unification variables and atoms are created dynamically by evaluating some and fresh expressions respectively (and may dynamically escape their lexical scopes).

As well as the usual let-binding and recursive function constructs which bind value identifiers, the some construct, $\mathsf{some}\, x : \sigma\, \mathsf{in}\, e$, binds free occurrences of the value identifier x in the expression e. The fresh atom generation construct $\mathsf{fresh}\, a : \alpha\, \mathsf{in}\, e$ binds all free occurrences of the atom a in e. As usual, substitution of MLSOS values for value identifiers is capture-avoiding: we identify MLSOS expressions up to α-conversion of bound value identifiers and atoms and write $e[v/x]$ for the capture-avoiding substitution of v for all free occurrences of x in e. These meta-level notions of α-equivalence and substitution for MLSOS should not be confused with the object-level notions that can be implemented in MLSOS. Object-level binding is represented with the $\langle\langle a \rangle\rangle(-)$ construct, but the latter is not itself a meta-level binding construct; for example, $\langle\langle a \rangle\rangle a$ and $\langle\langle b \rangle\rangle b$ are distinct MLSOS values when a and b are distinct atoms. The object-level substitution of

nominal terms for unification variables that is part of the MLSOS dynamics may involve capture of free atoms within the scope of the $\langle\langle a \rangle\rangle(-)$ construct. The only form of substitution needed for atoms is renaming via a permutation (see §1.4.1).

1.4.3 Constraints

We use a slightly richer language of constraints c than [23] to guide execution of MLSOS programs. They may be of the following forms:

$$\begin{array}{rcll} c & ::= & v =:= v' & \text{equality constraint,} \\ & | & a \# v & \text{freshness constraint,} \\ & | & v =/= v' & \textit{name} \text{ inequality constraint.} \end{array}$$

where v stands for a value (of some nominal arity σ). Equality constraints really mean equality of the appropriate α-equivalence classes, and a freshness constraint $a \# v$ expresses that a may not occur *free* in v. Note that, as emphasised by the typing rule for inequality constraints in Figure 1.6, these are only permitted between *names*, that is, atoms or suspensions of the same atom sort α. (Our name inequality constraints are similar to, but less general than, those described in [7, Chapter 7], since we place more restrictions on where suspensions may appear in the term syntax.) The only non-trivial cases are for inequalities between two suspensions: either

1. $\pi X =/= \pi' X'$, where X and X' are distinct unification variables—this constraint is left untouched as part of the "solution"; or
2. $\pi X =/= \pi' X$— this causes a finitely-wide branch, which tries to instantiate X with every atom a for which $\pi(a) \neq \pi'(a)$.

We can give a simple semantics to this constraint language in terms of instantiations of the unification variables contained in the constraints by *ground* values, that is, by ones not containing unification variables. Then, it can be shown that *satisfiability* of a finite set $\overline{c}$ of such constraints (which we write $\models \overline{c}$) is *decidable*, using an algorithm that extends nominal unification [23] to deal with name inequality constraints. This allows us, to a large extent, to factor constraint solving out of the operational semantics of the metalanguage, leading to a rather elegant presentation (see §1.4.4). In particular, the presentation does not need to use the meta-operation of substitution for unification variables, which is left implicit as part of constraint solving.

1.4.4 Operational semantics

Conceptually, evaluation of an MLSOS program consists of finitely many computation branches. During evaluation of a program, new branches may be spawned, each of which may return a result, or may fail finitely (which corresponds to where one would backtrack in Prolog), or may diverge. The operational semantics presented here is abstract in the sense that it does not specify any particular search strategy or treatment of failed computations. This is a succinct and elegant means

1. $И\overline{a}\ \exists\overline{X}\ (\overline{c};(S\circ(x.e))(v)) \longrightarrow_M И\overline{a}\ \exists\overline{X}\ (\overline{c};S(e[v/x]))$
2. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(\mathsf{let}\ x=e\ \mathsf{in}\ e')) \longrightarrow_M И\overline{a}\ \exists\overline{X}\ (\overline{c};(S\circ(x.e'))(e))$
3. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(v\ v')) \longrightarrow_M И\overline{a}\ \exists\overline{X}\ (\overline{c};S(e[v,v'/f,x]))$
   if $v=(\mathsf{fun}\ f(x:\tau):\tau'=e)$
4. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(c)) \longrightarrow_M И\overline{a}\ \exists\overline{X}\ ((\overline{c}\cup\{c\});S(\mathsf{yes}))$
   if $\models\overline{c}\cup\{c\}$
5. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(\mathsf{fresh}\ a:\alpha\ \mathsf{in}\ e)) \longrightarrow_M И\overline{a},a:\alpha\ \exists\overline{X}\ (\overline{c}';S(e))$
   if $a\notin dom(\overline{a})$ and $\overline{c}'\triangleq\{a\ \#\ X\mid X\in dom(\overline{X})\}\cup\overline{c}$
6. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(\mathsf{some}\ x:\sigma\ \mathsf{in}\ e)) \longrightarrow_M И\overline{a}\ \exists\overline{X},X:\sigma\ (\overline{c};S(e[\iota X/x]))$
   if $X\notin dom(\overline{X})$
7. $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(e_1\ \mathsf{or}\ e_2)) \longrightarrow_M И\overline{a}\ \exists\overline{X}\ (\overline{c};S(e_i))$
   where $i\in\{1,2\}$

FIGURE 1.5. $\longrightarrow_M$ Transition Rules

of developing the theory of the language, but is obviously not intended as the basis of an implementation.

The operational semantics takes the form of a *non-deterministic* single-step transition relation $\longrightarrow_M$ between abstract machine configurations C, defined in Figure 1.5. Our configurations make use of *frame-stacks*, S, which are either empty (Id) or of the form $S\circ(x.e)$, in which any free occurrences of the value identifier x in the expression e (which is on top of the stack) become bound. The frame-stack $S\circ(x.e)$ encodes an evaluation context where the value (if any) resulting from evaluation in the current evaluation position will be substituted for all free occurrences of x in e and the result evaluated in the context S. Non-empty frame-stacks are built up by let-bindings appearing in the program (see rule 2 in Figure 1.5, which reflects the fact that MLSOS is an *eager* language). Configurations are of the form $И\overline{a}\ \exists\overline{X}\ (\overline{c};S(e))$, where S is a frame-stack, e is the expression currently being evaluated, and $\overline{c}$ is a finite set of constraints which serve to guide proof-search. Here $\overline{a}$ and $\overline{X}$ are finite environments of sorting information about the atoms and unification variables appearing in the configuration. (The quantifier symbols $И$ and $\exists$ appearing in a configuration are merely punctuation, but suggest expected logical behaviour.)

Rule 5 in Figure 1.5 implements fresh atom generation. As we identify expressions up to α-renaming of atoms, the "freshness" side-condition of this rule ($a\notin dom(\overline{a})$) can always be satisfied. In this rule, the constraint set is updated with new freshness constraints between the freshly-generated atom and all unification variables that have been generated so far. Similarly, rule 6 generates a new unification variable and binds it to a value identifier. (In the suspension $\iota X,\iota$

$$\frac{a \in dom(\Gamma) \qquad \Gamma \vdash v : \sigma}{\Gamma \vdash a \,\#\, v : \mathsf{ans}} \qquad \frac{\Gamma \vdash v_1 : \sigma \qquad \Gamma \vdash v_2 : \sigma}{\Gamma \vdash v_1 \,=\!:=\, v_2 : \mathsf{ans}}$$

$$\frac{\Gamma \vdash v_1 : \alpha \qquad \Gamma \vdash v_2 : \alpha}{\Gamma \vdash v_1 \,=\!/=\, v_2 : \mathsf{ans}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma(X) = \sigma}{\Gamma \vdash \pi X : \sigma} \qquad \frac{\Gamma(a) = \alpha}{\Gamma \vdash a : \alpha}$$

$$\frac{\Gamma(a) = \alpha \qquad \Gamma \vdash v : \sigma}{\Gamma \vdash \langle\langle a \rangle\rangle v : \langle\langle \alpha \rangle\rangle \sigma} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{}{\Gamma \vdash \mathsf{yes} : \mathsf{ans}}$$

$$\frac{\Gamma \vdash v_1 : \sigma_1 \qquad \cdots \qquad \Gamma \vdash v_n : \sigma_n}{\Gamma \vdash (v_1, \ldots, v_n) : \sigma_1 * \cdots * \sigma_n} \qquad \frac{(\mathsf{K} : \sigma \to \delta) \in \Sigma \qquad \Gamma \vdash v : \sigma}{\Gamma \vdash \mathsf{K}\, v : \delta}$$

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau' \qquad f, x \notin dom(\Gamma)}{\Gamma \vdash \mathsf{fun}\, f(x : \tau) : \tau' = e : \tau \to \tau'} \qquad \frac{\Gamma \vdash v_1 : \tau \to \tau' \qquad \Gamma \vdash v_2 : \tau}{\Gamma \vdash v_1\, v_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau' \qquad x \notin dom(\Gamma)}{\Gamma \vdash \mathsf{let}\, x = e\, \mathsf{in}\, e' : \tau'}$$

$$\frac{\Gamma, a : \alpha \vdash e : \tau \qquad a \notin dom(\Gamma)}{\Gamma \vdash \mathsf{fresh}\, a : \alpha\, \mathsf{in}\, e : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau \qquad x \notin dom(\Gamma)}{\Gamma \vdash \mathsf{some}\, x : \sigma\, \mathsf{in}\, e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, \mathsf{or}\, e_2 : \tau}$$

FIGURE 1.6. Typing Rules For Core MLSOS: Constraints, Values And Expressions

denotes the identity permutation.) Rule 7 is a non-deterministic branch.

The rules defining the MLSOS type system are quite standard and are presented in Figure 1.6. Constraint expressions (§1.4.3) are assigned type ans. We say that a configuration $C = \mathcal{N}\overline{a}\ \exists\overline{X}\ (\overline{c}; S(e))$ is *well-typed* at type τ and write $\vdash C : \tau$ iff the sorting information contained in $\overline{a}$ and $\overline{X}$ is sufficient to infer that $\overline{c}$ is a finite set of well-formed constraints of type ans, and to assign types $\tau' \to \tau$ and τ' to the frame-stack S and expression e respectively (for some τ'). We also say that C is *satisfiable* if its constituent set of constraints is satisfiable, that is, if $\models \overline{c}$ holds (see §1.4.3). The following results (whose proofs are omitted) show that the MLSOS type system ensures for well-typed configurations that the only possibility for a computation branch to get stuck is when an unsatisfiable constraint is encountered.

Theorem 1.1 (Type preservation). *For all configurations C, C' and for all types*

τ, *if* $\vdash C : \tau$ *and* $C \longrightarrow_M C'$ *then* $\vdash C' : \tau$. *Furthermore, C is* satisfiable *if and only if* C' *is.*

Theorem 1.2 (Progress). *For all configurations C and for all types* τ, *if* $\vdash C : \tau$, *then either*

1. *C is of the form* $\forall\overline{a}\ \exists\overline{X}\ (\overline{c}; Id(v))$, *i.e. this branch has* terminated, *or*
2. *there exists a configuration* C' *such that* $C \longrightarrow_M C'$ *holds, or*
3. *C is* not satisfiable *(i.e. C is of the form* $\forall\overline{a}\ \exists\overline{X}\ (\overline{c}; S(c))$, *where i.e.* $\models \overline{c} \cup \{c\}$ *does not hold).*

Proof. These proofs are both by case analysis on C. □

1.5 RELATED WORK

FreshML [19, 21] is the immediate ancestor of MLSOS. It provides support for functional programming with binders up to α-equivalence, but without the proof-search facilities of MLSOS. Object-language names are represented by atoms, which are generated freshly when required (and never named directly). An abstraction $\langle\langle x \rangle\rangle e$ is deconstructed using *generative unbinding* [16]—a fresh name y is generated and *swapped* for x throughout e.

In FreshML, names may be represented using normal value identifiers such as x because everything is ground, i.e. there are no unknown names present. Shinwell and Pitts [20, 16] prove that this approach leads to correct representation of object-language syntax up to α-equivalence (in the sense that terms representing objects in the same α-equivalence class are contextually equivalent). FreshML can be thought of as a ground subset of MLSOS, in that FreshML does not include unification variables or proof-search facilities at the language level. In FreshML, the first clause in the definition of the capture-avoiding substitution function from Figure 1.2 can be implemented using a *name equality test* (syntax `if` $x = y$ `then` t `else` t'), since x and y are always concrete atoms. Our work aims to discover whether the FreshML-style treatment of binders can be successfully extended to a language with unification variables representing unknown terms.

FreshML is an *impure* functional language, in that the generation of atoms is an observable side-effect. The earlier FreshML-2000 language [15] had a *freshness inference* system for statically rejecting programs where freshly generated names were returned unabstracted (and hence were observable as a side-effect). This was dropped from the version of FreshML in [21], because it rejected too many reasonable-looking programs. However, recent work by Pottier [17] describes a tractable and practical decision procedure for rejecting impure programs in a FreshML-like language with user-supplied freshness assertions. That paper employs a system of *binding specifications* which is richer than our nominal signatures. These originate from work on Cαml [18], a tool which auto-generates Objective Caml code from such a binding specification.

```
1   func subst (var, lam, lam) = lam.
2   subst (X, E, (Var X)) = E.
3   subst (X, E, (Var Y)) = (Var Y) :- X # Y.
4   subst (X, E, App (E1, E2)) = App (subst (X, E, E1), subst (X, E, E2)).
5   subst (X, E, (Lam ⟨⟨a⟩⟩E')) = Lam ⟨⟨a⟩⟩(subst (X, E, E')) :- a # (X, E).

6   pred beta (lam, lam).
7   beta (E, E).
8   beta ((Lam ⟨⟨a⟩⟩E), (Lam ⟨⟨a⟩⟩E')) :- beta (E, E').
9   beta (App (E1, E2), App (E1', E2')) :- beta (E1, E1'), beta (E2, E2').
10  beta (App ((Lam ⟨⟨a⟩⟩E1), E2), E3) :-
        beta (E1, E1'), beta (E2, E2'), E3 = subst (a, E2', E1').
```

FIGURE 1.7. αProlog Parallel Reduction Program

The language most closely related to MLSOS in terms of functionality is Cheney and Urban's αProlog [5, 7]. This is a logic programming language based on *nominal logic* [14], which uses nominal unification to perform back-chaining.

Figure 1.7 presents a parallel reduction program in αProlog that mirrors the MLSOS code from Figure 1.2. For consistency we have adopted the MLSOS syntax for abstractions ($\langle\langle a\rangle\rangle E$ as opposed to $a\backslash E$ from [5]), and re-ordered the arguments to agree with the MLSOS program from Figure 1.2. In the base case of capture-avoiding substitution (Figure 1.7, lines 2-3) the αProlog program uses two clauses to implement the name equality test, and the freshness constraint $X \# Y$ corresponds to our name inequality constraint. In the λ case of capture-avoiding substitution (Figure 1.7, line 5) note that a freshness side-condition ($a \# (X, E)$) is necessary. This is because the atom a has not been generated freshly in the pattern, and hence the system cannot guarantee that a does not occur free in X or E unless the user makes this explicit. This highlights a difference between αProlog and MLSOS concerning the interpretation of the syntax a for an object-language name. Although both programming languages make use of nominal unification, in MLSOS the metavariable a stands for an atom to be generated freshly at runtime, whereas in αProlog it stands for *one particular atom* from the countably infinite set of atoms.

We would like to abstract away from the internal implementation of object-language names and binding as much as possible. In MLSOS we cannot write programs whose meaning depends upon particular atoms and the behaviour of MLSOS programs does not depend on which concrete atom is chosen to implement a particular object-language bound name. This relates to the *equivariance* property of nominal logic [14]—Cheney notes that *"because of equivariance, resolution based on nominal unification is incomplete for nominal logic"* [5]. That paper proposes replacing nominal unification by equivariant unification, in order to achieve completeness with respect to nominal logic, at the cost of **NP**-

completeness [6]. An alternative is to impose a syntactic criterion on αProlog programs which restricts to a subset of nominal logic formulae for which nominal unification is complete [8]. This is related to the problem discussed in §1.6.1.

Of course, there are alternative techniques for encoding binders, such as higher-order abstract syntax, which uses metalanguage binders to model object-language binders, and nameless de-Bruijn representations. The relative merits of the "nominal" techniques used in MLSOS and αProlog compared to such representations have been discussed elsewhere (see [23] for a survey). Broadly speaking, implementations using nameless representations are not very readable and can be inefficient, whereas higher-order abstract syntax systems such as Twelf [13] and Bedwyr [2] make it hard to use what is often the natural style of "nominal" programming in which concrete bound names are manipulated. They also suffer from a kind of incompleteness (due to their restriction to higher order pattern unification) similar to that discussed in §1.6.1.

Systems such as PLT Redex [12], which were designed specifically for the purpose of producing step-by-step reduction tools from a description of a language semantics, clearly bear comparison to MLSOS. A downside of PLT Redex is that it does not seem to provide automated support for α-equivalence of object-level binders, in the same way as MLSOS. Furthermore, in PLT Redex one is restricted to the operation of reducing a subterm in place, whereas in MLSOS one can write more liberal programs, for example to find all inhabitants of a particular type. However, PLT Redex does have a graphical visualisation toolkit.

There also exist mature, high-performance, general-purpose functional logic programming languages such as Curry [11] and Mercury [22]. However, these also lack built-in support for binders and α-equivalence and hence they are not such an attractive choice for the kind of applications we are targeting.

1.6 FUTURE WORK AND CONCLUSIONS

We have outlined some of the design decisions and motivations behind the development of MLSOS, a metalanguage designed for prototyping structural operational semantics definitions. In this section we mention some directions for future work.

1.6.1 Badly-behaved inductive definitions

MLSOS takes a "nominal" approach to expressing object-language binding syntax. As we have seen, it adopts a hybrid approach to implementing names in an object-language—they are represented by unification variables wherever possible, and by fresh atoms wherever essential (that is, whenever they appear in binding position). One might hope that any rule-based inductive definition could be implemented in MLSOS (along the lines of the example in §1.3) in a way that is complete—in the sense that MLSOS computes all and only correct solutions to user queries about the definition. (We hope that this notion of completeness is intuitively clear—we defer a formalisation to a future paper for reasons of space.)

However, there exist certain inductive definitions whose natural encoding in MLSOS is not complete. The same phenomenon occurs in αProlog (§1.5) and we can adapt to MLSOS an example from [9, Example 5.3]. Consider the set $\mathcal{S}$ defined by the single inference rule

$$\frac{}{R\ (x, t, \lambda x.t)}$$

which gives the graph of λ-abstraction on (α-equivalence classes) of untyped λ-terms. For example, the term $(a, a, \lambda b.b)$ is manifestly in $\mathcal{S}$, since $\lambda a.a$ and $\lambda b.b$ are α-equivalent. However, the natural MLSOS encoding of $\mathcal{S}$ as a function of type $\mathsf{var} \star \mathsf{lam} \star \mathsf{lam} \to \mathsf{ans}$ (using the nominal signature from §1.3), which we omit here for reasons of space, will fail to compute this solution. This is because the bound name x in the pattern $\lambda x.t$ in the conclusion of the rule must be modelled by a freshly-generated atom; however this name also appears free in the conclusion and hence its identity matters to the semantics of the rule. Thus, it seems that generating fresh atoms for bound names during pattern-matching prevents us from computing all members of $\mathcal{S}$.

The problem with this and similar definitions is that it allows us to inspect the identity of a name that appears in binding position by unbinding without any freshening. We would ideally like to rule out such badly-behaved definitions, as they violate the assumption that we can represent a bound name with *any* freshly-chosen atom. We aim to find some restricted class of inductively defined sets of α-equivalence classes of nominal terms for which MLSOS is powerful enough to give natural, yet complete encodings. Cheney and Urban [9, §5.2] consider such a restriction for the language αProlog which is discussed in §1.5, but it does not seem immediately applicable to MLSOS.

1.6.2 Correctness properties

We conjecture that the dynamics of the language (§1.4.4) is such that MLSOS representations of α-equivalent object-language terms are contextually equivalent. This form of correctness has been proved for FreshML [20, 16], but has yet to be established for MLSOS.

1.6.3 Conclusion

MLSOS provides a simple yet expressive medium for computing with abstract syntax trees identified up to α-conversion. A prototype implementation of the language exists, and a more efficient implementation is in the pipeline. This could be useful not only to programming language designers, but also as an educational tool for teaching operational semantics to students. The syntax and programming style should be familiar to people who are comfortable with functional, as opposed to logic, programming languages. MLSOS has the benefit of the full expressive power of functional programming—we have only scratched the surface of what can be done in terms of optimising the code that one would naïvely write. It is

possible that further refinements to the system, such as mode and determinism annotations, could allow us to automatically generate reasonable implementations of systems with minimal input from the user, and provide some degree of verification of certain metatheoretic properties of their definitions.

ACKNOWLEDGEMENTS

This work was supported by UK EPSRC grant EP/D000459/1.

REFERENCES

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanised metatheory for the masses: The POPLmark challenge. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

[2] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. Submitted to CADE 2007.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[4] Christophe Calvès and Maribel Fernández. Implementing nominal unification. In *3rd Int. Workshop on Term Graph Rewriting (TERMGRAPH'06), Vienna*, Electronic Notes in Theoretical Computer Science, 2006.

[5] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. 20th Int. Conf. on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, 2004.

[6] James Cheney. The complexity of equivariant unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *LNCS*, pages 332–344. Springer-Verlag, 2004.

[7] James Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, August 2004.

[8] James Cheney. Equivariant unification. In *Proceedings of the 2005 Conference on Rewriting Techniques and Applications (RTA 2005)*, number 3467 in LNCS, pages 74–89, 2005.

[9] James Cheney and Christian Urban. Nominal logic programming. Preprint available from `http://arxiv.org/abs/cs.PL/0609062`, 2006.

[10] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 39(4):502–514, 2004.

[11] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[12] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA) 2004*, 2004.

[13] F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Proceedings of the 16th Conference on Automated Deduction (CADE 1999)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, July 1999.

[14] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[15] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[16] A. M. Pitts and M. R. Shinwell. Generative unbinding of names. In *34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007), Nice, France*, pages 85–95. ACM Press, January 2007.

[17] F. Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, Wroclaw, Poland, July 2007. To appear.

[18] François Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52, March 2006.

[19] M. R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, Cambridge University, 2006.

[20] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[21] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[22] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Log. Program.*, 29(1-3):17–64, 1996.

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