

# FreshML: Programming with Binders Made Simple

28 March 2003

Mark R. Shinwell

Andrew M. Pitts

Murdoch J. Gabbay

Cambridge University Computer Laboratory, Cambridge, CB3 0FD, UK

## Abstract

FreshML extends ML with elegant and practical constructs for declaring and manipulating syntactical data involving binding operations. User-declared FreshML datatypes involving binders are concrete, in the sense that values of these types can be deconstructed by matching against patterns naming bound variables explicitly. Such matching may have a computational effect in which bound names get swapped with freshly generated names. Previous work on FreshML used a complicated static type system inferring information about the ‘freshness’ of names for expressions in order to tame such effects. The main contribution of this paper is to show (perhaps surprisingly) that a much simpler type system without freshness inference, coupled with name swapping and a conventional treatment of fresh name generation, suffices for FreshML’s crucial correctness property—namely that values of datatypes involving binders are operationally equivalent if and only if they represent  $\alpha$ -equivalent pieces of object-level syntax. This correctness result is established via a novel denotational semantics. FreshML without static freshness inference is no more impure than ML and our experiences programming in it show that it supports a programming style pleasingly close to informal practice when it comes to dealing with syntax modulo  $\alpha$ -equivalence.

## 1 Introduction

The facilities for user-declared datatypes and for deconstructing data by matching against patterns to be found in functional programming languages like ML and Haskell greatly simplify one of the main tasks for which these languages were originally intended, namely *metaprogramming*—the construction and manipulation of expressions in formal languages. A specification of an object-level language by a signature of syntactic categories and syntax-forming constructions of various arities, translates easily into meta-level programming language declarations of recursive datatypes whose values are in bijection with the parse trees of expressions of the corresponding syntactic categories. The declaration of recursive functions for manipulating the parse trees is made much simpler (and hence less error prone) through the use of patterns to match against parts of trees.

Unfortunately there is a pervasive problem that spoils this rosy picture. Formal languages very often involve binding operations; in which case, symbolic computations at the meta-level on object-level language expressions often only make sense, or at least only have good properties, when we operate not on parse trees themselves, but on equivalence classes of them for a relation of  $\alpha$ -equivalence identifying trees differing only in the names of bound entities. At the moment programmers have to deal with this issue case-by-case according to the nature of the object-level language

being implemented, using a self-imposed discipline. For example, they might work out (not so hard) and then correctly use (much harder) some ‘nameless’ representation of  $\alpha$ -equivalence classes of parse trees in the style of de Bruijn [8] for the particular object language in question.

The tedious and error-prone nature of *ad hoc* solutions to this semantically trivial, but pragmatically non-trivial issue of  $\alpha$ -equivalence is widely acknowledged; so there is a clear need for better automatic support for matters to do with object-level  $\alpha$ -equivalence in metaprogramming languages. FreshML is an experimental, ML-like language that provides such support. Its design was introduced by the second two authors in [25] and subsequently refined and implemented by the first author [27]. It provides the user with a general-purpose type construction, written  $\langle bty \rangle ty$ , for binding names (of various, user-defined types  $bty$ ) in expressions of arbitrary type  $ty$ . This type constructor for binding can be used in datatype declarations of types of object-level syntax to make the specification of the binding aspects of object-level languages purely declarative. For example, suppose we want to represent expressions of a small fragment of ML with the following forms:

<code>fn <math>x \Rightarrow e</math></code>	function abstraction
<code><math>e_1 e_2</math></code>	function application
<code>let val <math>x = e_1</math> in <math>e_2</math> end</code>	local value
<code>let fun <math>f x = e_1</math> in <math>e_2</math> end</code>	local recursive function

In FreshML we can declare a new type *name* of bindable names for object-level value identifiers

```
bindable_type name
```

and then declare a datatype *expr* for the above ML expressions:

```
datatype expr = Vid of name
              | Fn of <name>expr
              | App of expr * expr
              | Let of expr * <name>expr
              | Letfun of
                <name>((<name>expr) * expr)
```

The types we have used in this declaration tell the system exactly which data constructors are binders and in which way their arguments are bound. For example in an ML expression of the form `let fun  $f x = e_1$  in  $e_2$  end` there is a binding occurrence of the value identifier  $f$  whose scope is both of  $e_1$  and  $e_2$ ; and a binding occurrence of the value identifier  $x$  whose scope is just  $e_1$ . These binding scopes are reflected by the argument type of the corresponding constructor *Letfun* in the declaration of *expr*. Making the specification of binding structure declarative in this way would not be so useful were it not for the fact that the design of FreshML endows datatypes such as *expr* with two crucial properties:

**Abstractness:** when represented as values of the datatype, object-level expressions are operationally equivalent in FreshML if and only if they are  $\alpha$ -equivalent in the object language.

Indeed, datatypes like *expr* are *equality types* in the ML sense, and meta-level equality correctly represents object-level  $\alpha$ -equivalence.

**Concreteness:** values of such datatypes can be deconstructed by matching against patterns naming bound entities explicitly.

Indeed, FreshML provides automatic, language-level support for using the common informal idiom in which  $\alpha$ -equivalence classes are referred to via representative parse trees, with bound names changed ‘on the fly’ to make them distinct among themselves and distinct from any other names in the current context of use. For example, the following FreshML declaration of a function *subst* : *name*  $\rightarrow$  *expr*  $\rightarrow$  *expr*  $\rightarrow$  *expr* suffices for *subst* *x* *e*<sub>1</sub> *e*<sub>2</sub> to compute (a representation of) the ML expression obtained by capture-avoiding substitution of the expression represented by *e*<sub>1</sub> for all free occurrences of the value identifier named *x* in the expression represented by *e*<sub>2</sub>.

```
fun subst x e (Vid y) =
  if x # y then Vid y else e
| subst x e (Fn (<y>e1)) =
  Fn (<y>(subst x e e1))
| subst x e (App (e1, e2)) =
  App (subst x e e1, subst x e e2)
| subst x e (Let (e1, <y>e2)) =
  Let (subst x e e1, <y>(subst x e e2))
| subst x e (Letfun (<f>(<y>e1, e2))) =
  Letfun (<f>(<y>(subst x e e1), subst x e e2))
```

The interesting thing about the declaration of *subst* is that it is so simple!—only the first line of the declaration is not ‘boiler plate’ in the sense of [14]. In particular, in the clauses of the declaration that deal with substitution under a binder, it is enough to say what the result should be in the case when the bound name is sufficiently fresh. As we see in Section 2, FreshML ensures that this is the only case that arises when using the clause during evaluation; and this suffices for *subst* to determine a totally defined function because, by the results of Sect. 5, values of type *expr* represent  $\alpha$ -equivalence classes of expressions in the ML fragment.

As a language for ‘programming modulo  $\alpha$ -equivalence’, the Abstractness property says that FreshML is *correct*, whereas the Concreteness property says FreshML is *expressive*. Clearly these two properties are somewhat in opposition to each other and it is quite tricky to get them to co-exist. We were guided to the design in [25] that achieves this by using the mathematical model of binding in terms of name-swapping described in [10, 12, 24]. This provides a notion of the *support* of a mathematical object that specialises to the set of free names in case the object is an  $\alpha$ -equivalence class of parse trees involving binders. We put forward the following informal thesis relating what is sometimes called the ‘Barendregt variable convention’ [2, page 26] to this notion of support.

**Thesis:** when people carry out constructions on  $\alpha$ -equivalence classes of parse trees via representative trees using dynamically freshened bound names, the end result is well-defined, i.e. independent of which fresh bound names are chosen, because *the freshly chosen names do not occur in the support of the final result*.

The type system described in [25] enforces this italicised condi-

tion at compile-time by deducing information about a relation of *freshness* of names for expressions which is a decidable approximation to the (in general undecidable) ‘not-in-the-support-of’ relation. This results in a *pure* functional programming language, in as much as the static freshness inference ensures that the dynamics of replacing insufficiently fresh names on the fly is referentially transparent. Although purity is desirable (since it makes for better laws about program properties), our means of achieving it through static freshness inference turns out to have drawbacks. Since freshness is only a decidable approximation to the semantic notion of ‘not-in-the-support-of’, it is inevitable that the type-checker will reject some algorithms that do in fact conform to the above Thesis. Our experience with FreshML suggests that this happens uncomfortably often. We discuss why in Section 6. The main contribution of this paper is to show, perhaps surprisingly, that static freshness inference is not necessary for the crucial Abstractness property to hold in the presence of the Concreteness property, so long as one changes the operational semantics of FreshML as given in [25] to make fresh bindable names *generative* in the same way that some other sorts of names (references, exceptions and type names) are in ML. That this is so is not at all obvious and requires proof, which we give.

## Summary

We define a new version of FreshML, called *FreshML-Lite* (Sect. 2), with a considerably simpler type system than in [25], but a slightly more ‘effective’ dynamics. As explained in Sect. 3, the essence of the dynamics is the combination of generative names with *name-swapping*. To prove that the FreshML-Lite type system controls the dynamics sufficiently for object-level  $\alpha$ -equivalence to coincide with meta-level operational equivalence (the Abstractness Property mentioned above), in Sect. 4 we describe a denotational semantics of FreshML-Lite in the *FM-sets* model of binding [12] that gave rise to the design of FreshML in the first place. The denotational semantics is structured using a monad and makes use of *FM-cpos*, which seem interesting in their own right. Section 5 contains the main technical contributions of the paper: we prove that this denotational semantics is computationally adequate for the operational semantics (the proof uses a suitable logical relation between semantics and syntax) and then we use this result to establish the Abstractness Property (Theorem 5.6). Section 6 discusses the pros and cons of using static freshness inference as originally envisioned in [25] and Section 7 describes our experiences implementing FreshML and FreshML-Lite. From the programmer’s point of view, we claim that FreshML-Lite extends ML with elegant and practical constructs for declaring and manipulating syntactical data involving binding operations; the paper gives a few examples (capture-avoiding substitution, computation of free variables (Fig. 5) and normalisation by evaluation (Fig. 6)); more programming examples can be found at <http://www.freshml.org/>. Finally, Sect. 8 discusses related work and draws some conclusions about the results presented here and their implications for future work.

## 2 FreshML-Lite

This section gives the new definition of FreshML. Compared with [25, 27], we do without freshness inference in the type system and use a generative operational semantics for fresh names. To make the presentation more accessible, we use a cut-down language, which we call *FreshML-Lite*, combining the principal novelties of FreshML with a pure functional subset of ML, but without any polymorphism (either conventional ML-style polymorphism, or polymorphism over types of bindable names—see [27, Sect. 3.10] for FreshML’s treatment of the latter).

<b>Types</b>	$\tau \in \text{Ty}$	::=	<b>Values</b>	$v \in \text{Val}$	::=
bindable name	<code>name</code>		atom	$a$	
abstraction type	$\langle \text{name} \rangle \tau$		abstraction	$\langle a \rangle v$	
unit type	<code>unit</code>		pair	$(v, v)$	
product type	$\tau \times \tau$		unit	$()$	
function type	$\tau \rightarrow \tau$		closure	$[E, f(x) = e]$	
data type	$\delta$		constructed	$C$	
				$C v$	
<b>Patterns</b>	$pat$	::=			
variable	$x$				
abstraction	$\langle x \rangle x$				
pair	$(x, x)$				
<b>Declarations</b>	$dec$	::=			
value	<code>val</code> $pat = e$				
fresh atom	<code>fresh</code> $x$				
recursive function	<code>fun</code> $x (x) = e$				
sequential	$dec\ dec$				
<b>Expressions</b>	$e$	::=			
value identifier	$x$				
constructor	$C$				
atom inequality test	$e \# e$				
atom swapping	<code>swap</code> $e, e$ in $e$				
atom abstraction	$\langle e \rangle e$				
unit	$()$				
pair	$(e, e)$				
application	$e e$				
case split	<code>case</code> $e$ of $(C\ x \Rightarrow e \mid \dots)$				
local declaration	<code>let</code> $dec$ in $e$ end				
<b>Environments</b>	<b>Typing contexts</b>	<b>States</b>			
$E \in \text{VId} \xrightarrow{\text{fin}} \text{Val}$	$\Gamma \in \text{VId} \xrightarrow{\text{fin}} \text{Ty}$	$\bar{a} \in \mathbb{P}_{\text{fin}} \mathbb{A}$			
<b>Top-level datatype declaration</b>					
datatype $bool$	$= true \mid false$				
and	$\delta_1 = C \text{ of } \tau \mid \dots$				
	$\vdots$				

Figure 1. FreshML-Lite syntax and semantic objects

FreshML-Lite syntax is specified in Fig. 1. For simplicity we assume a single, top-level datatype declaration, including a type  $bool$  of booleans and possibly some other mutually recursively defined datatypes  $\delta_1, \delta_2, \dots$ . There is a single type of bindable names, called `name`, whose values are called *atoms*<sup>1</sup> and which are the elements of a fixed, countably infinite set  $\mathbb{A}$ . The type  $\langle \text{name} \rangle \tau$  of *atom-abstractions* has values given by pairs, written  $\langle a \rangle v$  and consisting of an atom  $a \in \mathbb{A}$  and a value  $v$  of type  $\tau$ . As we see below, FreshML-Lite’s semantics (both operational and denotational) identifies atom-abstraction values up to renaming the  $\langle \ \rangle$ -enclosed atom; for example,  $\langle a \rangle a$  and  $\langle a' \rangle a'$  turn out to be operationally equivalent values of type  $\langle \text{name} \rangle \text{name}$ . In FreshML-Lite programs, values of type `name` are introduced via local declarations of fresh atoms, in much the same way that names of references are introduced in ML; values of atom-abstraction type are introduced with expressions of the form  $\langle e_1 \rangle e_2$  and eliminated via a local declaration with an atom-abstraction pattern, `let val`  $\langle x \rangle y = e$  in  $e'$  end. For simplicity, Fig. 1 does not define nested patterns.

FreshML-Lite’s type system is quite standard compared with the one given in [25] (the latter infers freshness information in addi-

<sup>1</sup>The name stems from the denotational model given in Sect. 4.

## Expressions

$$\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name}}{\Gamma \vdash e_1 \# e_2 : \text{bool}} \quad (1)$$

$$\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \text{name} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{swap } e_1, e_2 \text{ in } e_3 : \tau} \quad (2)$$

$$\frac{\Gamma \vdash e_1 : \text{name} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \langle e_1 \rangle e_2 : \langle \text{name} \rangle \tau} \quad (3)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad (4)$$

$$\frac{\Gamma \vdash dec : \Gamma' \quad \Gamma + \Gamma' \vdash e : \tau}{\Gamma \vdash \text{let } dec \text{ in } e \text{ end} : \tau} \quad (5)$$

## Declarations

$$\frac{\Gamma \vdash e : \langle \text{name} \rangle \tau}{\Gamma \vdash \text{val } \langle x \rangle y = e : \{x \mapsto \text{name}, y \mapsto \tau\}} \quad (6)$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{val } (x, y) = e : \{x \mapsto \tau_1, y \mapsto \tau_2\}} \quad (7)$$

$$\frac{}{\Gamma \vdash \text{fresh } x : \{x \mapsto \text{name}\}} \quad (8)$$

## Values

$$\frac{a \in \mathbb{A}}{\vdash a : \text{name}} \quad (9)$$

$$\frac{}{\vdash () : \text{unit}} \quad (10)$$

$$\frac{a \in \mathbb{A} \quad \vdash v : \tau}{\vdash \langle a \rangle v : \langle \text{name} \rangle \tau} \quad (11)$$

$$\frac{\vdash v_1 : \tau_1 \quad \vdash v_2 : \tau_2}{\vdash (v_1, v_2) : \tau_1 \times \tau_2} \quad (12)$$

Figure 2. FreshML-Lite typing (excerpt)

tion to conventional ML typing properties). Figure 2 gives a selection of the typing rules. Comparing rules (3), (6) and (11) with rules (4), (7) and (12) respectively, we see that as far as typing properties are concerned,  $\langle \text{name} \rangle \tau$  is like a product type  $\text{name} \times \tau$  (cf. Theorem 3.2). However, the two types have different dynamic properties. Dynamics is specified as in the Definition of Standard ML [18], using an inductively defined evaluation relation of the form

$$\bar{a}, E \vdash e \Downarrow v, \bar{a}' \quad (22)$$

where the *states*  $\bar{a}$  and  $\bar{a}'$  are finite subsets of  $\mathbb{A}$  and the *value environment*  $E$  is a finite function mapping value identifiers to values. Figure 3 gives a selection of the evaluation rules. Rules (15) and (19) make use of the notation

$$(a\ a') \cdot v \quad (23)$$

which stands for the value obtained from  $v$  by swapping all occurrences of the atoms  $a$  and  $a'$  in it.

The state may grow during evaluation: if (22) holds, Lemma 2.2 below shows that  $\bar{a} \subseteq \bar{a}'$ . The features that cause it to grow are not

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow a, \bar{a}''}{\bar{a}, E \vdash e_1 \# e_2 \Downarrow \text{false}, \bar{a}''} \quad (13)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow a', \bar{a}'' \quad a \neq a'}{\bar{a}, E \vdash e_1 \# e_2 \Downarrow \text{true}, \bar{a}''} \quad (14)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a_1, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow a_2, \bar{a}''}{\bar{a}, E \vdash \text{swap } e_1, e_2 \text{ in } e_3 \Downarrow (a_1 a_2) \cdot v, \bar{a}'''} \quad (15)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow v, \bar{a}''}{\bar{a}, E \vdash \langle e_1 \rangle e_2 \Downarrow \langle a \rangle v, \bar{a}''} \quad (16)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow v_1, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow v_2, \bar{a}''}{\bar{a}, E \vdash (e_1, e_2) \Downarrow (v_1, v_2), \bar{a}''} \quad (17)$$

$$\frac{\bar{a}, E \vdash \text{dec} \Downarrow E', \bar{a}' \quad \bar{a}', E + E' \vdash e \Downarrow v, \bar{a}''}{\bar{a}, E \vdash \text{let } \text{dec} \text{ in } e \text{ end} \Downarrow v, \bar{a}''} \quad (18)$$

$$\frac{\bar{a}, E \vdash e \Downarrow \langle a \rangle v, \bar{a}' \quad a' \in \mathbb{A} - \bar{a} \quad v' = (a a') \cdot v}{\bar{a}, E \vdash \text{val } \langle x_1 \rangle x_2 = e \Downarrow \{x_1 \mapsto a', x_2 \mapsto v'\}, \bar{a}' \cup \{a'\}} \quad (19)$$

$$\frac{\bar{a}, E \vdash e \Downarrow (v_1, v_2), \bar{a}'}{\bar{a}, E \vdash \text{val } (x_1, x_2) = e \Downarrow \{x_1 \mapsto v_1, x_2 \mapsto v_2\}, \bar{a}'} \quad (20)$$

$$\frac{a \in \mathbb{A} - \bar{a}}{\bar{a}, E \vdash \text{fresh } x \Downarrow \{x \mapsto a\}, \bar{a} \cup \{a\}} \quad (21)$$

Figure 3. FreshML-Lite evaluation (excerpt)

only declarations of fresh atoms (see rule (21)), but also value declarations involving atom-abstraction patterns: see rule (19). This rule lies at the heart of our treatment of binders. It says that when matching an atom-abstraction pattern  $\langle x \rangle y$  against an atom-abstraction value  $\langle a \rangle v$ , we associate  $x$  not with  $a$ , but rather with a new atom  $a'$  not in the current state; and then  $y$  is associated with  $(a a') \cdot v$ . In fact the valid instances of the evaluation relation (22) all have the property that  $\bar{a}'$  contains all the atoms occurring in  $v$  (provided  $\bar{a}$  contains all those occurring in  $E$ , which we will always assume to be the case). So the new atom  $a'$  in rule (19) does not occur in  $v$  and therefore the swapping  $(a a') \cdot v$  is equal to  $[a := a'] \cdot v$ , the result of replacing all occurrences of  $a$  in  $v$  with  $a'$ . We postpone to Sect. 7 discussing ways of making the implementation of this rule more efficient, both by avoiding the generation of fresh atoms where possible and by delaying the computation of atom swaps (replacements) until needed.

We conclude this section by giving some ‘sanity checks’ on the typing and evaluation rules.

**Lemma 2.1.** Given value and typing environments  $E$  and  $\Gamma$ , write  $\vdash E : \Gamma$  to mean that  $\text{dom}(E) = \text{dom}(\Gamma)$  and that for each  $x \in \text{dom}(\Gamma)$ ,  $\vdash E(x) : \Gamma(x)$  holds. Then if  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$ ,  $\vdash E : \Gamma$  and  $\Gamma \vdash e : \tau$  hold, so does  $\vdash v : \tau$ .  $\square$

**Lemma 2.2.** Given a state  $\bar{a}$  containing all the atoms occurring in a value environment  $E$ , if  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$  holds then  $\bar{a} \subseteq \bar{a}'$ . Moreover, the pair  $(v, \bar{a}')$  is uniquely determined up to permuting

the atoms in  $\bar{a}' - \bar{a}$ : if  $\bar{a}, E \vdash e \Downarrow v', \bar{a}''$  also holds, then there is a bijection  $\pi$  between  $\bar{a}' - \bar{a}$  and  $\bar{a}'' - \bar{a}$  such that  $v'$  is obtained from  $v$  by applying  $\pi$  to the atoms occurring in it.  $\square$

### 3 The essence of FreshML is swapping

Although one can replace the use of the swapping operation  $(a a') \cdot v$  with the renaming operation  $[a := a'] \cdot v$  in the crucial rule (19) of Fig. 3, nevertheless swapping rather than renaming is the fundamental operation underlying our treatment of fresh bindable names. This is because in general we have to consider these operations not just when  $a'$  does not occur in  $v$ . Non-fresh renaming has bad properties; for example if  $a, b$  and  $c$  are three distinct atoms, then  $\langle b \rangle a$  and  $\langle c \rangle a$  are equivalent values, but this equivalence is not respected by  $[a := b] \cdot (-)$  which sends the first to  $\langle b \rangle b$  and the second to the inequivalent value  $\langle c \rangle b$ . The problem is the familiar one of ‘capture’ of free names by binders and might be repaired by developing a theory of capture-avoiding renaming. But a much simpler solution is just to use swapping  $(a b) \cdot (-)$ , which because of its self-inverse nature has excellent properties, including preservation of  $\alpha$ -equivalence. There is a growing body of evidence that name-swapping, and more generally permutations of names are very useful for describing properties of syntax involving binders: see [5, 6, 11, 24, 30]. So we have made swapping explicit in FreshML-Lite with the syntax  $\text{swap } e_1, e_2 \text{ in } e_3$  whose typing and evaluation rules are (2) and (15) in Figs 2 and 3.<sup>2</sup> Here is an example of its use.

**Example 3.1.** If  $\tau$  is an equality type, then so is the abstraction type  $\langle \text{name} \rangle \tau$ . If  $eq : \tau \times \tau \rightarrow \text{bool}$  is the equality function for  $\tau$ , then the equality function  $eq_a : \langle \text{name} \rangle \tau \times \langle \text{name} \rangle \tau \rightarrow \text{bool}$  for the type  $\langle \text{name} \rangle \tau$  is given by

```

fun eq_a (p) =
  let val (z, z') = p
      val <x>y = z
      val <x'>y' = z'
      fresh n
  in
    eq (swap x, n in y, swap x', n in y')
  end

```

This should be compared with the characterisation of  $\alpha$ -equivalence via swapping given in [12, Proposition 2.2].  $\square$

As far as the dynamics of FreshML-Lite are concerned, name-swapping is the only thing that is being added to ML. To see this, consider the extension of Standard ML [18] with a primitive polymorphic function  $\text{swap} : \text{unit ref} \rightarrow \text{unit ref} \rightarrow \alpha \rightarrow \alpha$  for swapping addresses (the values of type  $\text{unit ref}$ ) in ML values. Then FreshML-Lite can be translated into this language as in Fig. 4.

**Theorem 3.2.** The above translation preserves and reflects the FreshML-Lite typing and evaluation relations:  $\Gamma \vdash e : \tau$  holds in FreshML-Lite iff  $\llbracket \Gamma \rrbracket_{\text{ML}} \vdash \llbracket e \rrbracket_{\text{ML}} : \llbracket \tau \rrbracket_{\text{ML}}$  holds in ML+swap; and  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$  holds in FreshML-Lite iff  $\bar{a}, \llbracket E \rrbracket_{\text{ML}} \vdash \llbracket e \rrbracket_{\text{ML}} \Downarrow \llbracket v \rrbracket_{\text{ML}}, \bar{a}'$  holds in ML+swap.  $\square$

Since the translation is compositional, this theorem implies that it gives a ‘computationally adequate’ interpretation of FreshML-Lite; in other words if phrases become contextually equivalent when translated into ML+swap, then they are already contextually equivalent in FreshML-Lite. (We give the precise definition of contextual equivalence in Definition 5.3.) However, the converse is far

<sup>2</sup>The currently implemented version of FreshML instead uses the operation of *concreting* an atom-abstraction; swapping can be defined in terms of this (see [27, Sect. 3.9]) and *vice versa*.

## Types

$$\begin{aligned} \llbracket \text{name} \rrbracket_{\text{ML}} &= \text{unit ref} \\ \llbracket \langle \text{name} \rangle \tau \rrbracket_{\text{ML}} &= \llbracket \text{name} \rrbracket_{\text{ML}} \times \llbracket \tau \rrbracket_{\text{ML}} \end{aligned}$$

## Expressions

$$\begin{aligned} \llbracket e_1 \# e_2 \rrbracket_{\text{ML}} &= \llbracket e_1 \rrbracket_{\text{ML}} \langle \rangle \llbracket e_2 \rrbracket_{\text{ML}} \\ \llbracket \text{swap } e_1, e_2 \text{ in } e_3 \rrbracket_{\text{ML}} &= \text{swap } \llbracket e_1 \rrbracket_{\text{ML}} \llbracket e_2 \rrbracket_{\text{ML}} \llbracket e_3 \rrbracket_{\text{ML}} \\ \llbracket \langle e_1 \rangle e_2 \rrbracket_{\text{ML}} &= (\llbracket e_1 \rrbracket_{\text{ML}}, \llbracket e_2 \rrbracket_{\text{ML}}) \\ \llbracket \text{val } \langle x_1 \rangle x_2 = e \rrbracket_{\text{ML}} &= \text{val } (x'_1, x'_2) = \llbracket e \rrbracket_{\text{ML}} ; \\ &\quad \text{val } x_1 = \text{ref } (); \\ &\quad \text{val } x_2 = \text{swap } x_1 \ x'_1 \ x'_2 \\ &\quad \text{(where } x'_1, x'_2 \text{ are fresh)} \\ \llbracket \text{fresh } x \rrbracket_{\text{ML}} &= \text{val } x = \text{ref } () \end{aligned}$$

## Values

$$\begin{aligned} \llbracket a \rrbracket_{\text{ML}} &= a \\ \llbracket \langle a \rangle v \rrbracket_{\text{ML}} &= (\llbracket a \rrbracket_{\text{ML}}, \llbracket v \rrbracket_{\text{ML}}) \end{aligned}$$

The rest of the translation is the identity; for example

$$\begin{aligned} \llbracket \tau_1 \times \tau_2 \rrbracket_{\text{ML}} &= \llbracket \tau_1 \rrbracket_{\text{ML}} \times \llbracket \tau_2 \rrbracket_{\text{ML}} \\ \llbracket \langle e_1, e_2 \rangle \rrbracket_{\text{ML}} &= (\llbracket e_1 \rrbracket_{\text{ML}}, \llbracket e_2 \rrbracket_{\text{ML}}) \\ \llbracket \text{val } (x_1, x_2) = e \rrbracket_{\text{ML}} &= \text{val } (x_1, x_2) = \llbracket e \rrbracket_{\text{ML}} \\ \llbracket \langle v_1, v_2 \rangle \rrbracket_{\text{ML}} &= (\llbracket v_1 \rrbracket_{\text{ML}}, \llbracket v_2 \rrbracket_{\text{ML}}), \end{aligned}$$

etc.

**Figure 4. Translating FreshML-Lite into ML+swap**

from true, that is, the translation is far from being ‘fully abstract’. This is because atom-abstraction values in FreshML-Lite get translated to (atom,value)-pairs in ML+swap, where we can discover the identity of the first component in a way that we shall prove in Sect. 5 is impossible in FreshML-Lite. For example, it follows from the results in Sect. 5 that the following two expressions of type  $(\langle \text{name} \rangle \text{name}) \times (\langle \text{name} \rangle \text{name})$  are contextually equivalent in FreshML-Lite.

```
let fresh x fresh y in (⟨x⟩x, ⟨y⟩y) end
let fresh x in (⟨x⟩x, ⟨x⟩x) end
```

Under the translation they become

```
let val x = ref () val y = ref () in ((x, x), (y, y)) end
let val x = ref () in ((x, x), (x, x)) end
```

which are not contextually equivalent in ML+swap, since the context `let val x = - in #1(#1(x)) = #1(#2(x)) end` distinguishes them.<sup>3</sup>

It might seem from these observations that we could better mimic FreshML’s atom-abstraction type  $(\langle \text{name} \rangle \tau)$  in ML+swap by using an abstract type in ML with underlying representation type given by  $\text{name} \times \tau$ . However, being abstract, when such types are used in datatype definitions to represent syntax involving binders, we would lose the Concreteness property mentioned in the Introduction, i.e. the ability to match against patterns involving atom-abstraction, which seems so convenient in practice.

## 4 Denotational semantics

In this section we explain the meaning of FreshML-Lite types and expressions by giving them a denotational semantics within the uni-

<sup>3</sup>We use # both as in Standard ML for record selectors and for the infix operation testing for atom inequality.

verse of *FM-sets*. This is the permutation model of set theory devised by Fraenkel and Mostowski in the 1930s that is shown in [12] to provide a syntax-independent mathematical model of fresh bindable names and  $\alpha$ -conversion, by expressing those concepts purely in terms of the operation of swapping names (or atoms, as they are called in this context). It was this mathematical setting that gave rise to the original FreshML design [25]. We saw in the previous section that the dynamics of FreshML-Lite’s treatment of bindable names reduces to two things: a supply of dynamically generated fresh names (provided by the ML type `unit ref`) and name-swapping (which we had to introduce as a primitive). So it is perhaps not surprising that a mathematical model of freshness and name-swapping can provide a denotational semantics of FreshML-Lite that fits its operational semantics well. The *computational adequacy* result of Theorem 5.4 shows that it does; from this we deduce results about the correctness of representation of object-level  $\alpha$ -equivalence.

What is an FM-set? Ordinary sets are members of a universe obtained by starting with nothing ( $\emptyset$ ), at each stage forming new sets by collecting together all the subsets of the sets we have already formed, and continuing this way for rather a long time (transfinitely!). The universe of FM-sets differs from this in just two respects. First, instead of starting with  $\emptyset$ , we start with a fixed, infinite collection  $\mathbb{A}$  of ‘atoms’ (so-called because they will be members of the universe that do not possess elements themselves). Secondly, at each stage, from all possible subsets we only take those that possess a *finite support*, which by definition is a finite set  $\bar{a}$  of atoms such that for all  $a, b \in \mathbb{A} - \bar{a}$ , the subset  $X$  in question is equal to  $\{(a\ b) \cdot x \mid x \in X\}$ ; here  $(a\ b) \cdot x$  denotes the object of the FM-sets universe obtained from the object  $x$  by swapping the atoms  $a$  and  $b$  (a swapping action on elements of the universe is built in by construction). It follows (although not obviously so: see [12, Proposition 3.4]) that every object  $x$  of the universe of FM-sets possesses a *smallest* finite support, written  $\text{supp}(x)$ .

When making set-theoretic constructions with FM-sets, the only restriction that has to be borne in mind is that to remain within the universe the construction must result in something that again possesses a finite support. The axiomatic development of [10] shows that nearly the full range of logical and set-theoretical constructs has this property. Really the only thing excluded is the ability to form a set consisting of an *arbitrary choice* of infinitely many objects of the universe. For example, and this will be relevant below, if  $x_n$  (for  $n = 0, 1, \dots$ ) is a countable sequence of elements of an FM-set, although each element of the sequence possesses a finite support, there may be no single finite set of atoms that is a support for all the  $x_n$  simultaneously; only if there is such a finite set will the function sending each  $n$  to  $x_n$ , i.e. the set of ordered pairs  $\{(n, x_n) \mid n = 0, 1, \dots\}$ , be finitely supported and hence be in the universe of FM-sets.

FreshML-Lite features fixpoint recursion both at the level of types and of expressions. It is well known how to use domain theory to give denotational semantics for such features. Here we use not conventional domain theory, but rather domain theory within the universe of FM-sets; this gives us access to the FM-set former for atom-abstractions [12, Sect. 5] to give meaning to FreshML-Lite’s abstraction types  $(\langle \text{name} \rangle \tau)$ . We only need to consider  $\omega$ -complete partial orders (cpo) and  $\omega$ -continuous functions within the FM-sets universe, rather than any more sophisticated notion of domain. These can be described concretely as follows.

**Definition 4.1.** An *FM-cpo*  $D$  consists of an FM-set equipped with a finitely supported subset  $\sqsubseteq_D \subseteq D \times D$  (often abbreviated to just  $\sqsubseteq$ ) which is a partial order and has the property that any  $\omega$ -chain  $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$  in  $D$  possesses a least upper bound *pro-*

vided the chain is finitely supported, i.e. provided there is a single finite set of atoms  $\bar{a}$  with  $\text{supp}(d_n) \subseteq \bar{a}$  for all  $n$ . We also assume that  $\text{supp}(D) = \text{supp}(\sqsubseteq_D) = \emptyset$  (i.e. that the definition of  $D$  and its partial order do not depend upon particular atoms). A *morphism* of FM-cpos  $f : D \rightarrow D'$  is a function from  $D$  to  $D'$  that preserves the partial order, preserves swapping (i.e.  $f((a b) \cdot d) = (a b) \cdot f(d)$ ) and preserves least upper bounds of finitely supported  $\omega$ -chains.  $\square$

Thus an FM-cpo  $D$  does not possess least upper bounds of arbitrary  $\omega$ -chains. For example, the denotation of the FreshML-Lite type `name`  $\rightarrow$  `unit` turns out to be isomorphic to the FM-cpo of all subsets of  $\mathbb{A}$  that are either finite or whose complement is finite, partially ordered by inclusion; if we enumerate the elements of  $\mathbb{A} = \{a_0, a_1, a_2, \dots\}$ ,<sup>4</sup> then the chain  $\emptyset \sqsubseteq \{a_0\} \sqsubseteq \{a_0, a_2\} \sqsubseteq \{a_0, a_2, a_4\} \sqsubseteq \dots$  is not finitely supported and moreover does not possess a least upper bound in this FM-cpo.

The denotational semantics for FreshML-Lite using FM-cpos that we give is *monadic* in the sense of Moggi [19]. Each FreshML-Lite type  $\tau$  is assigned an FM-cpo  $\llbracket \tau \rrbracket$ ; (closed) expressions of type  $\tau$  get interpreted as elements of the FM-cpo  $\mathbb{T}[\llbracket \tau \rrbracket]$ , where  $\mathbb{T}$  is a particular ‘dynamic allocation’ monad on FM-cpos; whereas the elements of  $\llbracket \tau \rrbracket$  itself are used to interpret FreshML-Lite *values*,  $\vdash v : \tau$ . We give the precise definition of  $\mathbb{T}$  below. First, here is the definition of  $\llbracket \tau \rrbracket$  for the various FreshML-Lite types  $\tau$ .

- $\llbracket \text{unit} \rrbracket \stackrel{\text{def}}{=} \{1\}$ ,  $\llbracket \text{name} \rrbracket \stackrel{\text{def}}{=} \mathbb{A}$

Each FM-set  $X$  determines a *discrete* FM-cpo by taking  $\sqsubseteq$  to be the equality relation for  $X$ . We use a one-element set and the set of atoms  $\mathbb{A}$  as discrete FM-cpos to interpret `unit` and `name` respectively.

- $\llbracket \delta_n \rrbracket \stackrel{\text{def}}{=} D_n$

For datatypes we use FM-cpos  $D_n$  that are the minimal invariant solutions to the set of simultaneous recursive domain equations corresponding to the top-level mutually recursive datatype declaration. It is indeed possible to transfer to the setting of FM-cpos the usual technique of constructing minimal solutions to such equations using colimits of embedding-projection pairs (see [1, Sect. 5] for example); we omit the details.

- $\llbracket \langle \text{name} \rangle \tau \rrbracket \stackrel{\text{def}}{=} [\mathbb{A}][\llbracket \tau \rrbracket]$

For abstraction types we use the *atom-abstraction FM-cpo*  $[\mathbb{A}]D$  built from an FM-cpo  $D$  (generalising to FM-cpos the construction on FM-sets given in [12, Sect. 5]). This consists of equivalence classes for the pre-order (i.e. reflexive-transitive relation) on  $\mathbb{A} \times D$  given by:  $(a_1, d_1) \sqsubseteq (a_2, d_2)$  iff  $(a_1 b) \cdot d_1 \sqsubseteq_D (a_2 b) \cdot d_2$  holds for some (and indeed any)  $b \in \mathbb{A}$  such that  $b \notin \text{supp}(d_1) \cup \text{supp}(d_2) \cup \{a_1, a_2\}$ . Overloading the notation, we write  $[a]d$  for the element of  $[\mathbb{A}]D$  given by the equivalence class of the pair  $(a, d)$ . One can calculate that  $\text{supp}([a]d) = \text{supp}(d) - \{a\}$ .

- $\llbracket \tau \times \tau' \rrbracket \stackrel{\text{def}}{=} \llbracket \tau \rrbracket \times \llbracket \tau' \rrbracket$

For product types we use the *product* of FM-cpos, which as for ordinary cpos is just given by the set of all ordered pairs, partially ordered componentwise.

- $\llbracket \tau \rightarrow \tau' \rrbracket \stackrel{\text{def}}{=} \llbracket \tau \rrbracket \rightarrow \mathbb{T}[\llbracket \tau' \rrbracket]$

The *function FM-cpo*  $D \rightarrow D'$  formed from two FM-cpos  $D$  and  $D'$  consists of all finitely supported subsets of  $D \times D'$  that are total, monotone functions from  $D$  to  $D'$  preserving least upper bounds of finitely supported chains. For interpreting function types, we

<sup>4</sup>Any such bijection of the set of atoms with the natural numbers is external to the FM-sets universe since it cannot be finitely supported.

combine this construct with the dynamic allocation monad  $\mathbb{T}$  mentioned above. Thus values of type  $\tau \rightarrow \tau'$ , i.e. recursive function closures, are modelled by functions mapping values (elements of  $\llbracket \tau \rrbracket$ ) to ‘computations’ of values (elements of  $\mathbb{T}[\llbracket \tau' \rrbracket]$ ). The definition of the monad  $\mathbb{T}$  reflects the fact that evaluation of expressions in FreshML-Lite either diverges ( $\mathbb{T}[\llbracket \tau \rrbracket]$  contains a least element  $\perp$  for modelling this), or returns a value that may involve finitely many fresh atoms ( $\mathbb{T}[\llbracket \tau \rrbracket]$  contains elements of the form  $\nu \bar{a} \cdot d$  with  $\bar{a} \in \mathbb{P}_{\text{fin}} \mathbb{A}$  and  $d \in D$  for modelling this).  $\mathbb{T}D$  is constructed by quotienting  $(\mathbb{P}_{\text{fin}} \mathbb{A} \times D)_{\perp}$  by a suitable equivalence relation, mimicking one of the dynamic allocation monads on a presheaf category used by Stark to model the Pitts-Stark  $\nu$ -calculus [28]. We omit the details here, and just note that  $\mathbb{T}D$  carries the following structure.

- A (mono)morphism  $\eta : D \rightarrow \mathbb{T}D$  (this is the unit of the monad).
- A least element  $\perp \in \mathbb{T}D$ .
- A morphism  $\nu : [\mathbb{A}]D \rightarrow D$  (‘restriction’) satisfying for all  $a, b \in \mathbb{A}$  and  $x \in \mathbb{T}D$  that

$$\nu[a](\nu[b]x) = \nu[b](\nu[a]x) \quad (24)$$

$$\nu[a]x = x \text{ if } a \notin \text{supp}(x). \quad (25)$$

$\mathbb{T}D$  is determined up to isomorphism by the fact that it is minimal with these properties, i.e. it has a certain category-theoretic universal property (whose statement we omit). The universal property can be used to define the lifting part of the monad structure from that given above and to verify the usual monad laws. Each non-bottom element of  $\mathbb{T}D$  can be presented as

$$\nu \bar{a} \cdot d \stackrel{\text{def}}{=} \nu[a_1] \dots \nu[a_n] \eta(d) \quad (26)$$

where  $\bar{a} = \{a_1, \dots, a_n\}$  is a finite (possibly empty) set of atoms occurring in the support of  $d \in D$  (by property (24), the order in which we list the elements of  $\bar{a}$  on the right-hand side of (26) is immaterial). Use of this monad satisfying the laws (24) and (25), rather than a more simple one which just pairs up values and states, turns out to be necessary in order to prove the correctness result of Theorem 5.6.

Returning to the definition of the denotational semantics of FreshML-Lite, we now have the machinery to construct the denotations of values, declarations and expressions in the following form:

- if  $\Gamma \vdash \text{dec} : \Gamma'$  is derivable, then  $\llbracket \Gamma \vdash \text{dec} : \Gamma' \rrbracket : [\Gamma] \rightarrow \mathbb{T}[\Gamma']$ ;
- if  $\Gamma \vdash e : \tau$  is derivable, then  $\llbracket \Gamma \vdash e : \tau \rrbracket : [\Gamma] \rightarrow \mathbb{T}[\llbracket \tau \rrbracket]$ ;
- if  $\vdash v : \tau$  is derivable, then  $\llbracket \vdash v : \tau \rrbracket \in \llbracket \tau \rrbracket$

where the denotation  $\llbracket \Gamma \rrbracket$  of a typing environment  $\Gamma$  is given by the product of the FM-cpos  $\llbracket \Gamma(x) \rrbracket$  as  $x$  ranges over the finite domain of definition of  $\Gamma$ . The definition proceeds by induction on the derivation of the judgements; we just give the interesting clauses of the definition.

**Fresh name declaration:**  $\Gamma \vdash \text{fresh } x : \Gamma'$

$\llbracket \Gamma \vdash \text{fresh } x : \Gamma' \rrbracket(\rho) \stackrel{\text{def}}{=} \nu\{a\} \cdot \{x \mapsto a\}$ , for some/any  $a \in \mathbb{A}$  (by construction of  $\mathbb{T}$ , the right-hand side is independent of  $a$ ).

**Value declaration for abstractions:**  $\Gamma \vdash \text{val } \langle x \rangle x' = e : \Gamma'$

Let  $d = \llbracket \Gamma \vdash e : \langle \text{name} \rangle \tau \rrbracket(\rho)$ . Then, for  $m$  as defined below,

$$\llbracket \Gamma \vdash \text{val } \langle x \rangle x' = e : \Gamma' \rrbracket(\rho) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } d = \perp \\ m & \text{otherwise.} \end{cases}$$

For  $d \neq \perp$ , we have that  $d = \nu \bar{a} \cdot [a]d'$  and we can define  $m \stackrel{\text{def}}{=} \nu(\{a'\} \uplus \bar{a}) \cdot \{x \mapsto a', x' \mapsto (a a') \cdot d'\}$  for some/any  $a' \notin \bar{a} \cup \{a\} \cup \text{supp}(d')$ .

**Swap expression:**  $\Gamma \vdash \text{swap } e_1, e_2 \text{ in } e_3 : \tau$   
 Let  $a_1 = \llbracket \Gamma \vdash e_1 : \text{name} \rrbracket(\rho)$ ,  $a_2 = \llbracket \Gamma \vdash e_2 : \text{name} \rrbracket(\rho)$  and  $d = \llbracket \Gamma \vdash e_3 : \tau \rrbracket(\rho)$ . Then

$$\llbracket \Gamma \vdash \text{swap } e_1, e_2 \text{ in } e_3 : \tau \rrbracket(\rho) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } a_1 = \perp, \\ & \text{or } a_2 = \perp \\ (a_1 a_2) \cdot d & \text{otherwise.} \end{cases}$$

**Abstraction expression:**  $\Gamma \vdash \langle e_1 \rangle e_2 : \langle \text{name} \rangle \tau$   
 Given  $d_1 = \llbracket \Gamma \vdash e_1 : \text{name} \rrbracket(\rho)$  and  $d_2 = \llbracket \Gamma \vdash e_2 : \tau \rrbracket(\rho)$ , then  $\llbracket \Gamma \vdash \langle e_1 \rangle e_2 : \langle \text{name} \rangle \tau \rrbracket(\rho) \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } d_1 = \perp, \text{ or } d_2 = \perp \\ \nu(\bar{a} \uplus \bar{a}') \cdot [a]d & \text{otherwise, where } d_1 = \nu \bar{a} \cdot a, \\ & d_2 = \nu \bar{a}' \cdot d \text{ and } \bar{a} \cap \bar{a}' = \emptyset. \end{cases}$

**Atom value:**  $\llbracket \vdash a : \text{name} \rrbracket \stackrel{\text{def}}{=} a$ .

**Abstraction value:**  $\llbracket \vdash \langle a \rangle v : \langle \text{name} \rangle \tau \rrbracket \stackrel{\text{def}}{=} [a] \llbracket \vdash v : \tau \rrbracket$ .

The places in the above definition where we refer to ‘some/any atom’ are really instances of the *freshness quantifier*  $\mathcal{V}$  introduced in [12]: we choose some atom satisfying the condition, but in fact any one satisfying the condition will do, since the atom does not occur in the support of the object being defined.

## 5 Correctness

In this section we show that the denotational semantics given in Sect. 4 fits the operational semantics of FreshML-Lite sufficiently closely for us to be able to prove that values of recursively defined FreshML-Lite datatypes can correctly represent  $\alpha$ -equivalence classes of object-level syntax involving binders (the Abstractness property mentioned in the Introduction). For simplicity we will use the familiar example of the untyped  $\lambda$ -calculus as the object-language to be represented in FreshML-Lite. However, the results we give for this language easily generalise to other signatures involving binders.

We noted in the previous section that the denotation  $\llbracket \delta \rrbracket$  of a declared FreshML-Lite datatype  $\delta$  is given by a recursively defined FM-cpo. When the declaration of  $\delta$  only involves product and abstraction types, but not function types (so that  $\delta$  is in fact an equality type, in the sense of ML), then  $\llbracket \delta \rrbracket$  is actually a discrete FM-cpo, given by an *inductively-defined FM-set*.

**Example 5.1.** Given the datatype declaration

$$\begin{aligned} \text{datatype } \text{lam} = & \text{Var of name} \\ & | \text{Lam of } \langle \text{name} \rangle \text{lam} \\ & | \text{App of } \text{lam} \times \text{lam} \end{aligned}$$

then  $\llbracket \text{lam} \rrbracket$  is isomorphic to the discrete FM-cpo given by the inductively defined FM-set  $\mu X. (\mathbb{A} + [\mathbb{A}]X + X \times X)$ . As shown in [12, Theorem 6.2], this FM-set is in bijection with the set  $\Lambda(\mathbb{A})/\equiv_\alpha$  of  $\alpha$ -equivalence classes  $[t]_{\equiv_\alpha}$  of untyped  $\lambda$ -terms  $t$  with variables drawn from  $\mathbb{A}$ :

$$t \in \Lambda(\mathbb{A}) ::= a \mid \lambda a. t \mid t t$$

(The swapping action is  $(a b) \cdot [t]_{\equiv_\alpha} = [(a b) \cdot t]_{\equiv_\alpha}$ , where as usual,  $(a b) \cdot t$  is the parse tree obtained from  $t$  by interchanging all occurrences of  $a$  and  $b$ ; and the support of  $[t]_{\equiv_\alpha}$  turns out to be the finite set of *free* variables of  $t$ .) It is not hard to see from the typing rules for values in FreshML-Lite that there is a bijection between

$\lambda$ -terms  $t \in \Lambda(\mathbb{A})$  and values  $(t)_v$  of type  $\text{lam}$ , given by

$$\begin{aligned} (a)_v & \stackrel{\text{def}}{=} \text{Var } a \\ (\lambda a. t)_v & \stackrel{\text{def}}{=} \text{Lam } (\langle a \rangle (t)_v) \\ (t_1 t_2)_v & \stackrel{\text{def}}{=} \text{App } ((t_1)_v, (t_2)_v). \end{aligned}$$

One can show by induction on the structure of  $t$  that under the isomorphism between  $\Lambda(\mathbb{A})/\equiv_\alpha$  and  $\llbracket \text{lam} \rrbracket$  mentioned above, the  $\alpha$ -equivalence class of  $t$  is identified with the denotation of  $(t)_v$ :  $[t]_{\equiv_\alpha} = \llbracket \vdash (t)_v : \text{lam} \rrbracket$ . Therefore for all  $\lambda$ -terms  $t$  and  $t'$  we have

$$t \equiv_\alpha t' \quad \text{iff} \quad \llbracket \vdash (t)_v : \text{lam} \rrbracket = \llbracket \vdash (t')_v : \text{lam} \rrbracket. \quad (27)$$

□

The upshot of this example is that denotations of FreshML-Lite values of datatypes like  $\text{lam}$  represent  $\alpha$ -equivalence classes of object-language terms in a one-to-one fashion. To connect this correctness property to FreshML-Lite programs and their operational behaviour we have to do two things. First, we have to examine the translation of object-level terms into expressions rather than values, since these are what the programmer writes; secondly, we have to relate equality of denotation of expressions to an appropriate notion of operational behaviour (Definition 5.3).

To tackle the first issue, let us continue with the running example of  $\lambda$ -terms as the object-language.

**Example 5.2.** The set of atoms  $\mathbb{A}$  and the set of value identifiers Vid are both countably infinite sets and hence in bijection. Let us fix some explicit bijection  $a_i \leftrightarrow x_i$  by enumerating each set. Then we translate the  $\lambda$ -terms  $t$  of Example 5.1 into FreshML-Lite expressions  $(t)_e$  as follows:

$$\begin{aligned} (a_i)_e & \stackrel{\text{def}}{=} \text{Var } x_i \\ (\lambda a_i. t)_e & \stackrel{\text{def}}{=} \text{let fresh } x_i \text{ in Lam } (\langle x_i \rangle (t)_e) \text{ end} \\ (t_1 t_2)_e & \stackrel{\text{def}}{=} \text{App } ((t_1)_e, (t_2)_e). \end{aligned}$$

One can show by induction on the structure of the  $\lambda$ -term  $t$  that if  $t$  has free variables among the finite set  $\{a_1, \dots, a_n\}$  and if  $\Gamma$  is the typing context mapping all the corresponding value identifiers  $x_1, \dots, x_n$  to  $\text{name}$ , then

- $\Gamma \vdash (t)_e : \text{lam}$  holds;
- for any  $\rho \in \llbracket \Gamma \rrbracket = \mathbb{A}^n$ ,  $\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \nu \emptyset \cdot \llbracket \vdash (\rho t)_v : \text{lam} \rrbracket \in \mathbb{T}[\llbracket \text{lam} \rrbracket]$ , where  $\rho t$  is the  $\lambda$ -term obtained by simultaneous capture-avoiding substitution of  $\rho(x_i)$  for  $a_i$  in  $t$  (for  $i = 1, \dots, n$ ).

(When it comes to the step for  $\lambda$ -abstractions in the proof of the second property, the ‘garbage collection’ property (25) of the monad  $\mathbb{T}$  is crucial, combined with the fact that in an atom-abstraction FM-cpo  $[\mathbb{A}]D$ , an element of the form  $[a]d$  never contains  $a$  in its support.) From this we get for all  $\lambda$ -terms  $t$  and  $t'$  that

$$t \equiv_\alpha t' \quad \text{iff} \quad \llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket \quad (28)$$

where  $\Gamma$  is as above. For if  $t \equiv_\alpha t'$ , then  $\rho t \equiv_\alpha \rho t'$  for any  $\rho$ ; so from (27) we get  $\llbracket \vdash (\rho t)_v : \text{lam} \rrbracket = \llbracket \vdash (\rho t')_v : \text{lam} \rrbracket$  and therefore  $\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket(\rho)$ ; since this holds for any  $\rho$ , we have the left-to-right implication in (28). Conversely, if  $\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket(\rho)$  holds for any  $\rho$ , from above we get  $\nu \emptyset \cdot \llbracket \vdash (\rho t)_v : \text{lam} \rrbracket = \nu \emptyset \cdot \llbracket \vdash (\rho t')_v : \text{lam} \rrbracket$  in  $\mathbb{T}[\llbracket \text{lam} \rrbracket]$ ; but this implies that  $\llbracket \vdash (\rho t)_v : \text{lam} \rrbracket = \llbracket \vdash (\rho t')_v : \text{lam} \rrbracket$  in  $\llbracket \text{lam} \rrbracket$  (since

the unit of the monad  $\mathbb{T}$  is a monomorphism); so by (27) we have  $\rho t \equiv_{\alpha} \rho t'$ ; and then we can take  $\rho$  to be  $\rho(x_i) = a_i$  ( $i = 1, \dots, n$ ), for which  $\rho t \equiv_{\alpha} t$  and  $\rho t' \equiv_{\alpha} t'$ , to conclude that  $t \equiv_{\alpha} t'$ .  $\square$

Turning to the second of the two issues mentioned above, namely relating equality of denotation of expressions to an appropriate notion of operational behaviour, we give a notion of *contextual equivalence* for FreshML-Lite expressions. Roughly speaking, two expressions (of the same type) are contextually equivalent if they are interchangeable in any complete program without changing the program's observable behaviour. We take programs to be closed expressions of type `unit` and their observable behaviour to be whether they evaluate to the unique value of that type, ignoring any fresh atoms that may be created along the way. (In fact for strict functional languages, many other choices of 'program' and 'observable behaviour' lead to the same relation of contextual equivalence.) Here is the precise definition, which refers to the typing and evaluation relations defined in Sect. 2. It also makes use of the usual notion of a *context*  $\mathcal{C}[-]$ , which is an expression from the grammar in Fig. 1 with a subexpression replaced by the placeholder '-'; and then  $\mathcal{C}[e]$  denotes the expression resulting from replacing '-' by  $e$ .

**Definition 5.3.** Given  $\Gamma \vdash e : \tau$  and  $\Gamma \vdash e' : \tau$ , we write

$$\Gamma \vdash e \approx_{\text{ctx}} e' : \tau$$

and say that  $e$  and  $e'$  are *contextually equivalent* if for all contexts  $\mathcal{C}[-]$  with  $\emptyset \vdash \mathcal{C}[e] : \text{unit}$  and  $\emptyset \vdash \mathcal{C}[e'] : \text{unit}$ , we have  $\exists \bar{a} (\emptyset, \emptyset \vdash \mathcal{C}[e] \Downarrow (\cdot), \bar{a})$  if and only if  $\exists \bar{a}' (\emptyset, \emptyset \vdash \mathcal{C}[e'] \Downarrow (\cdot), \bar{a}')$ .  $\square$

**Theorem 5.4. (Computational adequacy)** Suppose  $\Gamma \vdash e : \tau$ ,  $\vdash E : \Gamma$  and  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$  (with  $\bar{a}$  containing the atoms of  $E$ ). Let  $\rho$  be given by  $\rho(x) = \llbracket \vdash E(x) : \Gamma(x) \rrbracket$ , for  $x \in \text{dom}(\Gamma)$ .

- (a)  $\llbracket \Gamma \vdash e : \tau \rrbracket(\rho)$  is equal to the non-bottom element  $\nu(\bar{a}' - \bar{a}) \cdot \llbracket \vdash v : \tau \rrbracket$  of  $\mathbb{T}[\tau]$ .
- (b) Conversely, if  $\llbracket \Gamma \vdash e : \tau \rrbracket(\rho) \neq \perp$ , then  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$  holds for some  $v$  and  $\bar{a}'$ .

**PROOF SKETCH.** The proof of part (a) is by induction on the derivation of  $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$ . The proof of part (b) is more involved. We use a standard method for it, based on the construction of type-indexed *logical relations* relating domain elements to pieces of syntax. The construction of such logical relations is complicated by the fact that, as for ML, FreshML allows the use of function types in the declaration of recursively defined datatypes (see Fig. 6 in Sect. 6 for an example of this). This precludes the use of simple, inductive techniques; instead, we deduce the existence of the logical relations by using the theory of *minimal invariant relations* as developed in [23]. This provides a general framework which can be instantiated to the setting of FM-cpos to deduce that the logical relations we need do exist.  $\square$

Parts (a) and (b) of the theorem combine to give the first part of the following corollary. The second part can be deduced from the proof of the theorem, by exploiting particular properties of the logical relation at *equality types*, which for the simplified language given in Sect. 2 we can take to be types not involving use of the function type construct (either in themselves, or in the declarations of any datatypes that they involve).

**Corollary 5.5.** Suppose  $\Gamma \vdash e : \tau$  and  $\Gamma \vdash e' : \tau$ . If  $\llbracket \Gamma \vdash e : \tau \rrbracket = \llbracket \Gamma \vdash e' : \tau \rrbracket$ , then  $\Gamma \vdash e \approx_{\text{ctx}} e' : \tau$ . The converse holds if  $\tau$  is an equality type.  $\square$

**Theorem 5.6. (Correctness of representation)** Under the translation given in Example 5.2 of  $\lambda$ -terms  $t$  into FreshML-Lite expressions  $(t)_e$  of type `lam`,  $\alpha$ -equivalence of  $\lambda$ -terms corresponds to

```

1 fun remove(<x>[]) = []
2   | remove(<x>(y::ys)) =
3     if x # y then y::(remove(<x>ys))
4     else remove(<x>ys);
5 fun fv(Var x) = [x]
6   | fv(Lam(<x>t)) = remove(<x>(fv t))
7   | fv(App(t1,t2)) = (fv t1)@(fv t2);
8 fun is_closed t = ((fv t)=[])

```

**Figure 5. Testing for closed  $\lambda$ -terms**

contextual equivalence of FreshML-Lite expressions: given two  $\lambda$ -terms  $t$  and  $t'$ , with free variables among  $\{a_1, \dots, a_n\}$  say, letting  $\Gamma$  be the typing context that maps the corresponding value identifiers  $x_1, \dots, x_n$  to name, we have

$$t \equiv_{\alpha} t' \quad \text{iff} \quad \Gamma \vdash (t)_e \approx_{\text{ctx}} (t')_e : \text{lam}.$$

**PROOF.** The type `lam` is an equality type, so we can combine Corollary 5.5 with (28) to get the desired conclusion.  $\square$

Let us emphasise once again that although we have stated this correctness theorem for the simple object language of  $\lambda$ -terms, similar results hold for object languages specified by a general notion of binding signature (such as the 'nominal algebras' of [13]) and datatypes in FreshML-Lite whose declarations are derived from the signature in the simple declarative fashion discussed in the Introduction.

What more could one want from such a representation of object-languages in a metalanguage? Well, to be useful, the metalanguage should provide rich facilities for writing algorithms for manipulating representations of object-level terms. FreshML's ability to match on abstraction patterns is particularly useful in this respect. However, a basic facility we have yet to discuss is *the ability to recognise that a meta-level expression does represent some object-level term*. For impure functional programming languages such as ML, the ability to do this cannot just be a matter of typing and termination. For example, in a full version of FreshML, besides the non-termination caused by fixpoint recursion there are computational side-effects from references, exceptions, and so on; so there will always be closed expressions of the type `lam` from Example 5.1 that are not the representation of any  $\lambda$ -term via the encoding of Example 5.2. But even when we cut down to the FreshML-Lite language of Sect. 2, the generative nature of our treatment of fresh bindable names means that there are terminating closed expressions of type `lam` that are not contextually equivalent to  $(t)_e$  for any closed  $\lambda$ -term  $t$ . A typical example of such an expression is

$$\text{genvar} \stackrel{\text{def}}{=} \text{let fresh } x \text{ in } (\text{Var } x) \text{ end} \quad (29)$$

which evaluates to 'some fresh object-level variable' whenever we evaluate it. Once we identify  $\llbracket \text{lam} \rrbracket$  with  $\Lambda(\mathbb{A})/\equiv_{\alpha}$  as in Example 5.1, the denotational semantics of `genvar` is  $\llbracket \emptyset \vdash \text{genvar} : \text{lam} \rrbracket = \nu\{a\} \cdot (\llbracket a \rrbracket_{\equiv_{\alpha}}) \in \mathbb{T}[\llbracket \text{lam} \rrbracket]$  (and of course  $\llbracket a \rrbracket_{\equiv_{\alpha}}$  is just  $\{a\}$ ); and from this and the results of Sect. 5 it follows that, as claimed, `genvar` is not contextually equivalent to  $(t)_e$  for any closed  $\lambda$ -term  $t$ .

Nevertheless we can easily write FreshML-Lite boolean-valued functions for recognising whether expressions are the encoding of some object-level term. For example, the function `is_closed` declared in Fig. 5 is one way of doing the job for our example encoding of  $\lambda$ -terms (not the most efficient way perhaps—we are going for clarity over efficiency). Here for the sake of

legibility we have augmented the syntax of FreshML-Lite with nested patterns and ML’s usual syntax for lists. The function  $remove: (\text{name}) (\text{name list}) \rightarrow \text{name list}$  removes an abstracted name from the list of names within the abstraction; the function  $fv: \text{lam} \rightarrow \text{name list}$  computes the list of names of free variables of a  $\lambda$ -term (possibly with repeats); and then the function  $is\_closed: \text{lam} \rightarrow \text{bool}$  tests whether that list is empty and hence whether the term is closed. For example  $is\_closed\ genvar$  evaluates to `false` (creating a fresh atom as it does so):  $genvar$  is not a closed  $\lambda$ -term. More generally, we can use the results in this section and the previous one to show that for a closed FreshML-Lite expression  $e$  of type `lam`, if evaluation of  $e$  terminates (i.e. if  $\emptyset, \emptyset \vdash e \Downarrow v, \bar{a}$  holds for some  $v$  and  $\bar{a}$ ), then  $\emptyset \vdash is\_closed\ e \approx_{ctx} true : \text{bool}$  if and only if  $\emptyset \vdash e \approx_{ctx} (t)_e : \text{lam}$  holds for some closed  $\lambda$ -term  $t$ .

## 6 Freshness inference

Recall from Sect. 4 the notion of *finite support*, which is an important aspect of the FM-sets model. When doing pencil-and-paper constructions with  $\alpha$ -equivalence classes via representatives using dynamically freshened bound names (i.e. when using the ‘Barendregt variable convention’ [2, page 26]), it seems always to be the case that the end result is well-defined, i.e. independent of which fresh bound names are chosen, by virtue of the fact that the freshly chosen names do not occur in the support of the final result. For if  $a, b \notin \text{supp}(x)$ , then from the definition of  $\text{supp}$  we have that swapping  $a$  and  $b$  has no effect on  $x$ , i.e.  $(a\ b) \cdot x = x$ . (The definition of the denotational semantics at the end of Sect. 4 contains several examples of this phenomenon, for example.) This is the informal Thesis put forward in the Introduction.

The original design of FreshML in [25] attempted to enforce this property of freshly chosen names not being in the support of final results by deducing information at compile-time about the support of the denotation of expressions. As well as conventional type information, the type-checker builds up information about a relation of *freshness* between value identifiers and expressions,  $x \# e$ , that is a sound approximation to the ‘not-in-the-support-of’ relation  $a \notin \text{supp}(\llbracket e \rrbracket)$  in the denotational semantics. The freshness relation is decidable by design, and hence can only ever approximate the ‘not-in-the-support-of’ relation, which is undecidable for the usual recursion-theoretic reasons.

As an example of how freshness inference works, consider the definition of the function  $fv$  computing the list of free variables of a  $\lambda$ -term in Fig. 5. At line 6,  $x \# remove(\langle x \rangle (fv\ t))$  holds, because  $x$  is always fresh for an object of the form  $\langle x \rangle e$  (and also because  $x$  is fresh for  $remove$ , since the definition of that function does not depend upon any atom in particular). Therefore, when matching against this clause of the definition, the final result does not depend upon which particular fresh atom gets associated with the bound name  $x$ . And indeed FreshML can spot this at compile-time and allow  $fv$  as a well-typed expression of type  $\text{lam} \rightarrow \text{name list}$ . On the other hand,  $x$  is most definitely not fresh for the expression  $Var\ x$  (corresponding to the fact that  $a \in \{a\} = \text{supp}(a)$ ); and the original FreshML design will not admit the expression  $genvar$  defined in equation (29) as a well-typed expression of type  $\text{lam}$ .

As these examples indicate, static freshness inference influences which expressions are typeable—exactly the ones where dynamic allocation of fresh names causes no observable side-effects, as it turns out. Therefore the operational semantics in the original FreshML design could be simplified by not threading through evaluation a state containing the names generated so far. For this ‘pure’ form of the evaluation relation, freshness inference is definitely

needed for the kind of correctness properties for representing syntax modulo  $\alpha$ -equivalence that we discussed in Sect. 5. We have seen that a simple type system without freshness inference suffices for ‘no confusion’ correctness results such as Theorem 5.6, provided one also uses a more conventional operational semantics employing states. With freshness inference we also get ‘no junk’ results. For example, in the version of FreshML described in [25] (which had no other effects), every closed terminating expression of type  $\text{lam}$  is of the form  $(t)_e$  for some closed  $\lambda$ -term  $t$ . As we saw at the end of Sect. 5, although FreshML-Lite does not have this property, at least it can recognise dynamically that an expression represents a closed  $\lambda$ -term. So it turns out that the real effect of including static freshness-checking in the design is to give us a more pure functional language and hence one with better programming laws. However, as we explain next, our experiences both implementing and using freshness inference indicate that it is a rather inconvenient luxury.

As far as implementation goes, unfortunately the freshness inference algorithm has to proceed not solely on the structure of expressions, as for the typing of Standard ML for example, but also at certain points on the structure of (inferred) types. The necessity to perform type-directed inference is due to the important notion of *purity*—judgements which state that values of *certain types* (such as `bool`, `int`, or `string`, for example) can never contain atoms of a certain kind in their support. Purity inference is needed for the treatment of many seemingly-innocuous pieces of syntax. For example in order to deduce that the function expression  $\text{fn } \langle x \rangle y \Rightarrow y + 1$  respects the conditions imposed by freshness checking, it is necessary to deduce that  $x \# y + 1$  holds. In such cases one needs to examine the type of the match to attempt to verify such conditions. For this simple example that type is `int`, and the required purity judgement would state that a value of type `int` cannot ever contain any atoms in its support and hence that  $x \# y + 1$  does indeed hold. More complicated problems of this kind arise when one considers recursive datatype declarations; in order to deduce which kinds<sup>5</sup> of atoms may or may not occur in the support of values of the declared datatypes, it is necessary to converge on a fixed point in a similar manner to the procedure for maximising equality in ML [18, Sect. 4.9]. Having incorporated this procedure into the freshness inference algorithm more programs are typeable, but the results of type-checking get harder for the user to understand and predict.

Experience writing syntax-manipulating programs in the original version of FreshML with freshness inference was one of the main driving factors that led to the development of the simpler version of the language presented in this paper. We were too frequently forced to adopt an obtuse coding style in order for programs to pass the freshness checks, or prevented by the statics from writing certain programs at all. This is not due to the *concept* behind the inference procedures, but rather the fact that the decidable freshness inference implemented by the type checker is only ever going to be an approximation to the fundamentally undecidable problem of determining which atoms are in the support of a particular semantic value at run-time. Thus, as ever, there has to be a trade-off between the expressivity of the language and the properties of the programs that we are able to deduce statically. In particular, the static freshness relation is a particularly bad approximation to the semantic ‘not-in-the-support-of’ relation at function types, due to the logical complexity of the latter. (It turns out that for a function  $f$  in the FM-sets universe,  $a \notin \text{supp}(f)$  iff for all but finitely many atoms  $b$  it is the case that  $(\forall x \in \text{dom}(f)) (a\ b) \cdot (f\ x) = f((a\ b) \cdot x)$ .)

<sup>5</sup>FreshML permits the declaration of different types of bindable names, whose values range over disjoint copies of  $\mathbb{A}$ .

```

1 (* syntax *)
2 datatype lam =
3   | Var of name
4   | Lam of <name>lam
5   | App of lam*lam;
6 (* semantics *)
7 datatype sem =
8   | L of sem->sem (* function *)
9   | N of neu      (* neutral *)
10 and neu =
11   | V of name      (* free variable *)
12   | A of neu*sem; (* neutral application *)
13 (* reification reify:sem->lam *)
14 fun reify(L f) =
15   let fresh x:name
16   in Lam(<x>(reify(f(N(V x))))) end
17   | reify(N n) = reify n
18 and reify(V x) = Var x
19   | reify(A(n,d)) =
20     App(reify n, reify d);
21 (* evaluation eval:lam->sem *)
22 fun evals [] (Var x) = N(V x)
23   | evals ((x,t)::env) (Var y) =
24     if x # y then evals env (Var y)
25     else t
26   | evals env (Lam(<x>t)) =
27     L(fn d => evals((x,d)::env) t)
28   | evals env (App(t1,t2)) =
29     (case evals env t1 of
30      | L f => f(evals env t2)
31      | N n => N(A(n,evals env t2)));
32 fun eval t = evals [] t;
33 (* normalisation norm:lam->lam *)
34 fun norm t = reify(eval t)

```

Figure 6. Normalisation by evaluation

Figure 6 gives an interesting illustration of the shortcomings of static freshness inference, while at the same time showing off the ease with which highly non-trivial syntax-manipulating algorithms can be expressed in FreshML-Lite; the algorithm in question computes the normal form of an untyped  $\lambda$ -term (if any) using *normalisation by evaluation* [15, 4] in a form suggested to us by Thierry Coquand [private communication]. In FreshML-Lite (augmented with nested patterns and syntax for lists), the function *norm* has type  $lam \rightarrow lam$  (and does indeed compute normal forms where they exist); but in the original version of FreshML with freshness inference *norm* does not type check, because the helper function *evals* does not. The reason for this has to do with the clause in the definition of *evals* at lines 22–23, where the type system cannot deduce that  $x \# (\text{fn } d \Rightarrow \text{evals } ((x,d)::\text{env}) t)$  holds, under the assumption that  $x \# \text{evals}$  and  $x \# \text{env}$  hold. Indeed the proof of the corresponding property of supports in the denotational model, though true, is far from immediate.

We advocate the use of FreshML-Lite, which does not try to enforce the Introduction’s Thesis (the property of freshly chosen names not being in the support of final results) at compile-time. However, this does not mean that freshness inference is useless. It could be used in a program logic for verifying properties of FreshML programs. Furthermore, even if we do not use it to reject programs whose use of fresh names is not referentially transparent, information about freshness deduced at compile-time may be very useful for increasing efficiency of the run-time implementation. We turn to this issue

<b>Canonical values</b>	$c ::=$
atom	$a$
abstraction	$\langle a \rangle p$
pair	$(p, p)$
unit	$()$
closure	$[P, f(x) = e]$
constructed	$C$
	$C p$
<b>Non-canonical values</b>	$p ::= \pi \bullet c$
<b>Explicit permutations</b>	$\pi ::=$
identity	$\square$
composite	$(a a') :: \pi$
<b>Value environments</b>	$P ::= [x \mapsto p, \dots]$

(where  $a, a'$  range over  $\mathbb{A}$  and  $x, f$  over  $\text{VId}$ )

Figure 7. Values with delayed swapping

next.

## 7 Implementation

The source code of an experimental implementation of the FreshML language (written in Objective Caml) is available at the web site <http://www.freshml.org/>. Our implementation provides the Core of Standard ML [18] together with FreshML’s distinctive features for programming with binders. There is an interactive interpreter, together with support for processing FreshML code held in individual source files, but as yet no modules layer. The ‘Lite’ version of the language without static freshness inference, as described in this paper, is not yet documented at the web site. Nevertheless, it can be obtained by starting `freshml` with the command-line argument `--lite`. The web site contains examples of programming with binders in FreshML (including Barthe’s classification algorithm for type-checking injective Pure Type Systems [3]; and programs calculating the possible labelled transitions from a  $\pi$ -calculus [17] process, using various forms of encoding). We invite readers to try FreshML(-Lite) for themselves!

We saw in Sect. 3 that the dynamics of FreshML-Lite can be implemented by translating it into ML augmented with a primitive function  $\text{swap} : \text{unit ref} \rightarrow \text{unit ref} \rightarrow \alpha \rightarrow \alpha$  for swapping address names in ML values. One can add such a primitive function to existing ML implementations using unsafe features<sup>6</sup>; but since the representation of ML values in such systems was not designed with swapping in mind, the use of such a primitive can result in unacceptable amounts of copying. Instead we use a representation of FreshML values suggested by Mark Shields [private communication] using *delayed swapping*. In this scheme, shown in Fig. 7, values  $p$  are equipped at each structural level with ‘explicit permutations’ of atoms (represented by finite lists of pairs of atoms,  $\pi$ , standing for the sequential composition, reading from left to right, of the corresponding atom swaps). Evaluation produces values in *canonical form*,  $c$ , where the outermost constructor is manifest; but value environments  $P$  need only associate value identifiers with values that are not necessarily in canonical form. For example, evaluation rule (15) from Fig. 3 becomes

$$\frac{\bar{a}, P \vdash e_1 \Downarrow a_1, \bar{a}' \quad \bar{a}', P \vdash e_2 \Downarrow a_2, \bar{a}''}{\bar{a}'', P \vdash e_3 \Downarrow c, \bar{a}'''} \quad c' = \text{cf}((a_1 a_2) :: \square \bullet c) \quad (30)$$

$$\frac{}{\bar{a}, P \vdash \text{swap } e_1, e_2 \text{ in } e_3 \Downarrow c', \bar{a}'''}$$

<sup>6</sup>We are grateful to Claudio Russo for showing us how to do this in Moscow ML.

where  $p \mapsto \text{cf}(p)$  is an auxiliary function converting a non-canonical value to canonical form. This function just has to push the outermost explicit permutation through one structural level, applying it to any atom it meets; this is relatively inexpensive to implement compared with a traversal of the whole parse tree of a value to find and swap its atoms. In a typical FreshML program, it seems that the majority of swapping will arise from the deconstruction of atom-abstraction values  $\langle a \rangle v$ . In the delayed swapping implementation scheme, the evaluation rule for this (cf. rule (19) in Fig. 3) becomes

$$\frac{\bar{a}, P \vdash e \Downarrow \langle a \rangle (\pi \bullet c), \bar{a}' \quad a' \in \mathbb{A} - \bar{a}' \quad p_1 = [] \bullet a' \quad p_2 = (a a') :: \pi \bullet c}{\bar{a}, P \vdash \text{val} \langle x_1 \rangle x_2 = e \Downarrow \{x_1 \mapsto p_1, x_2 \mapsto p_2\}, \bar{a}' \cup \{a'\}} \quad (31)$$

with the swap  $(a a')$  simply appended to the list  $\pi$ . A further important optimisation is to avoid choosing a fresh atom  $a'$  in the above rule when at all possible: if it happens that  $a \in \mathbb{A} - \bar{a}'$ , then we can simplify this evaluation step by taking  $a' = a$  and replacing  $(a a') :: \pi \bullet c$  with  $\pi \bullet c$ . To make this possible more often, one can try to reduce the state (set of atoms) that results from an evaluation by removing from it any atoms not in the support of the returned value. For example, we can improve evaluation rule (19) in this respect by replacing it with

$$\frac{\bar{a}, P \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a} \cup \{a\}, P \vdash e_2 \Downarrow c, \bar{a}''}{\bar{a}, P \vdash \langle e_1 \rangle e_2 \Downarrow \langle a \rangle ([] \bullet c), \bar{a}'' - \{a\}} \quad (32)$$

since  $a$  is definitely not in the support of  $\langle a \rangle ([] \bullet c)$ . We have yet to fully exploit this kind of optimisation. Clearly one should be able to make use of the static freshness inference described in Sect. 6 to increase the effectiveness of such optimisations in the dynamics. For example, the notion of *purity* mentioned there would enable one simply to discard the explicit permutation from the outside of a non-canonical value whose type has been found to be pure when computing its canonical form. So although with FreshML-Lite we are advocating a less strict type system than in the original FreshML design, freshness inference may still be very useful for optimising the dynamics.

The results of Sect. 5 can be adapted to prove that the optimised operational semantics with delayed swapping is correct for the denotational semantics of Sect. 4. Alternatively, one might prove directly that the optimised version is equivalent to the original operational semantics given in Sect. 2, although we have yet to do that.

## 8 Related and future work

FreshML is unique among metaprogramming languages in providing language-wide support for object-level  $\alpha$ -equivalence while still allowing the user to refer to bound entities by name. Miller [16] proposed tackling the same problem domain by incorporating the techniques of *higher order abstract syntax*, HOAS [22], into an ML-like programming language,  $\text{ML}_\lambda$ , with intentional function types  $\text{ty} \Rightarrow \text{ty}'$ . Compared with HOAS, FreshML's underlying theory of binders [12] is less ambitious in what it seeks to lift to the metalevel: like HOAS it promotes object-level renaming to the metalevel (via the swapping operation), but unlike HOAS it leaves object-level substitution to be defined case-by-case using structural recursion. The advantage is that FreshML data types are concrete and their denotational semantics in the universe of FM-sets retain the pleasant recursion/induction properties of classical first-order algebraic data types: see [12, Sect. 6]. It is also the case that bound names are inaccessible to the  $\text{ML}_\lambda$  programmer, whereas they are accessible to a FreshML programmer in a controlled way. Part of that control is via the generation of fresh names that can

be tested for inequality. Use of this seemingly small computational effect (which in fact has rather subtle interactions with higher order functions: see [26]) of course has a long history in functional programming, from Lisp's *gensym* to more recent advocates, such as [21, 7]. However, combining it with name-swapping and controlling the two through the use of the abstraction type  $\langle bty \rangle ty$  is unique to FreshML. Since the original design was published in [25], the use of swapping and freshness to deal with  $\alpha$ -equivalence and name restriction has been taken up by others, such as in [6]. Indeed, it was reading this work that inspired us to consider removing the rather restrictive freshness checking from FreshML's statics and design the new version of the language presented here, FreshML-Lite. FreshML's approach to names and binding has also inspired work on open code types in homogeneous metaprogramming languages by Nanevski and Pfenning [20].

It is by no means obvious that FreshML-Lite should have the correctness properties established in Sect. 5. We had to work quite hard to establish them, introducing a new denotational model, FM-cpos, that we think is interesting in its own right. Traditional, Scott-Strachey models of dynamically allocated local names are not sufficiently abstract to establish these correctness results (roughly speaking, they do not verify the laws (24) and (25)). Therefore, beginning with Oles, Reynolds and Moggi, various people have developed and applied 'dynamic allocation' monads in categories of functors valued in  $\omega$ -cpos: see [9, 29] for example. Even though the details of Sects 4 and 5 may seem heavy, in fact they are far less so than the functor category approach—both conceptually (we are just doing traditional domain theory, but in a slightly different classical set theory) and practically (constructions on FM-cpos, especially function spaces, are much easier to describe concretely than are the analogous constructs in functor categories). We believe that FM-cpos should be investigated as an interesting model of restriction (in the sense of  $\pi$ -calculus) and spatial locality in general; the work in [5] already takes steps in this direction.

Simplifying FreshML to produce FreshML-Lite as we have done opens up many interesting possibilities. It opens the door to full-scale functional language implementations incorporating our approach to programming with binders. Currently, for example, the FreshML implementation does not include support for Standard ML modules, since we do not know how freshness inference interacts with module typing; but without freshness inference it seems straightforward, in principle, to incorporate our novel language constructs into the full Standard ML (or Objective Caml) and we plan to try to do so. Maybe more important is to find a way of incorporating some of our approach to programming with binders into existing ML systems, such as the O'Caml compiler; at the moment we cannot see how to do that in a lightweight way that reuses as much of the considerable O'Caml infrastructure as possible, but produces a reasonably efficient implementation of the new features.

An interesting language feature which is made practical to implement by the move away from freshness inference techniques is that of abstracting by more than just a single atom. Work is underway to permit the formation of values of 'arbitrary' abstraction type  $\langle \tau \rangle \tau'$  ( $\tau$  must be an equality type for reasons that we do not have space to go into here). With this one could bind a list, or a tree, of atoms in a value—and hence, for example, more easily represent the kind of binding involved in the ML match construct (which binds all the variables occurring in a pattern). It is not known how to perform the freshness inference procedure when types of this form are permitted; the main problem being that one needs to be able to judge which atoms *are* in the support of a value as well as which are not. FreshML-Lite neatly circumvents this problem.

The features we have described here for programming with binders seem really useful. Therefore the school of pure, lazy functional programming should have them too—there should be a FreshHaskell! Once again, the simplification afforded by the move to the ‘Lite’ version of FreshML holds out hope that this is possible. (The problem of getting effective static freshness information for functions in a strict language, becomes the same and very widespread problem for closures in a call-by-need language.) The effects of generating fresh names are left exposed in FreshML-Lite; in ‘FreshHaskell’ one would presumably encapsulate them using a monad, mimicking the use of a monad in the denotational semantics of Sect. 4. However, the design of ‘FreshHaskell’ remains to be investigated.

## 9 Acknowledgements

This research was funded by UK EPSRC grant GR/R07615/01 and by a donation from Microsoft’s Cambridge Research Laboratory. We thank Luca Cardelli, Thierry Coquand, Simon Peyton Jones, Claudio Russo, Mark Shields and Keith Wansbrough for helpful comments and feedback on this work. Peter White contributed much to the implementation work, which benefitted enormously from use of the OCaml compiler of INRIA’s *projet Cristal*.

## 10 References

- [1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 3. Semantic Structures*, chapter 1. Oxford University Press, 1994.
- [2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [3] G. Barthe. Type-checking injective pure type systems. *Journal of Functional Programming*, 9(6):675–698, 1999.
- [4] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed  $\lambda$ -calculus. In *LICS 1991*, pages 203–211. IEEE Computer Society Press, 1991.
- [5] L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *CONCUR 2002*, volume 2421 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, 2002.
- [6] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *FOSSACS 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2003.
- [7] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *ASIAN’99*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, 1999.
- [8] N. G. de Bruijn. 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] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the  $\pi$ -calculus (extended abstract). In *LICS 1996*, pages 43–54. IEEE Computer Society Press, 1996.
- [10] M. J. Gabbay. *A Theory of Inductive Definitions with  $\alpha$ -Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, University of Cambridge, 2000.
- [11] M. J. Gabbay. The pi-calculus in Fraenkel-Mostowski. Submitted, 2003.
- [12] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
- [13] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer-Verlag, 2001.
- [14] R. Laemmel and S. L. Peyton Jones. Scrap your boilerplate: A practical approach to generic programming. In *TLDI 2003*. ACM Press, 2003.
- [15] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium ’73*, pages 73–118. North-Holland, 1975.
- [16] D. A. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, 1990.
- [17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
- [18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [19] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- [20] A. Nanevski. Meta-programming with names and necessity. In *ICFP 2002*, pages 206–217. ACM Press, New York, 2002.
- [21] M. Odersky. A functional theory of local names. In *POPL 1994*, pages 48–59. ACM Press, 1994.
- [22] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI 1988*, pages 199–208. ACM Press, 1988.
- [23] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
- [24] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, to appear. (A preliminary version appeared in *TACS 2001*, LNCS 2215, Springer-Verlag, 2001, pp 219–242.)
- [25] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.
- [26] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What’s new? In *MFCS 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.
- [27] M. R. Shinwell and A. M. Pitts. *FreshML User Manual*. Cambridge University Computer Laboratory, November 2002. Available at <http://www.freshml.org/docs/>.
- [28] I. D. B. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.
- [29] I. D. B. Stark. A fully abstract domain model for the  $\pi$ -calculus. In *LICS 1996*, pages 36–42. IEEE Computer Society Press, 1996.
- [30] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. Preprint, January 2003.