

FreshML: Programming with Binders Made Simple

Mark R. Shinwell

Andrew M. Pitts

Murdoch J. Gabbay

Cambridge University Computer Laboratory, Cambridge, CB3 0FD, UK
{Mark.Shinwell,Andrew.Pitts,Murdoch.Gabbay}@cl.cam.ac.uk

Abstract

FreshML extends ML with elegant and practical constructs for declaring and manipulating syntactical data involving statically scoped 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. This may have the computational effect of swapping bound names with freshly generated ones; previous work on FreshML used a complicated static type system inferring information about the ‘freshness’ of names for expressions in order to tame this effect. The main contribution of this paper is to show (perhaps surprisingly) that a standard type system without freshness inference, coupled with a conventional treatment of fresh name generation, suffices for FreshML’s crucial correctness property that values of datatypes involving binders are operationally equivalent if and only if they represent α -equivalent pieces of object-level syntax. This is established via a novel denotational semantics. FreshML without static freshness inference is no more impure than ML and experience with it shows that it supports a programming style pleasingly close to informal practice when it comes to dealing with object-level syntax modulo α -equivalence.

Categories and Subject Descriptors

D.3.1 [Programming Languages]: Formal Definitions and Theory—*syntax*; D.3.2 [Programming Languages]: Language Classifications—*applicative (functional) languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*data types and structures, patterns*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*denotational semantics, operational semantics*

General Terms

Languages, Design, Theory

Keywords

Metaprogramming, variable binding, alpha-conversion

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP’03 August 25–29, 2003, Uppsala, Sweden.
Copyright 2003 ACM 1-58113-756-7/03/0008 ...\$5.00

1 Introduction

User-declared datatypes and pattern-matching in functional programming languages like ML and Haskell simplify one of the main tasks for which these languages were intended, namely *metaprogramming*—the construction and manipulation of syntactical structures. In particular, the declaration of recursive functions for manipulating the parse trees of object-level languages is made much simpler (and hence less error prone) through the use of patterns to match against parts of trees. Unfortunately a pervasive problem spoils this rosy picture: object-level languages often involve binding operations. In this case meta-level programs only make sense, or at least only have good properties, when we operate not on parse trees but on their equivalence classes for a relation of α -equivalence identifying trees differing only in the names of bound entities. At the moment programmers deal with this 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 α -equivalence classes of parse trees in the style of de Bruijn [8]. The tedious and error-prone nature of *ad hoc* solutions to this semantically trivial, but pragmatically non-trivial issue of α -equivalence is widely acknowledged [27, Sect.13]. We need better automatic support for object-level α -equivalence in metaprogramming languages.

FreshML is an ML-like language which provides such support. Its design was introduced by Pitts and Gabbay [25] and subsequently refined and implemented by Shinwell [29]. It provides the user with a general-purpose type construction, written $\langle bty \rangle ty$, for binding names of user-declared type bty in expressions of arbitrary type ty . This can be used in datatype declarations of types representing object-level syntax to specify information about binding in the type. For example, suppose we want to represent expressions of a small fragment of ML with the following forms:

x	value identifier
$\text{fn } x \Rightarrow e$	function abstraction
$e_1 e_2$	function application
$\text{let fun } f x = e_1 \text{ in } e_2 \text{ end}$	local recursive function

In FreshML we can declare a new type $name$ of bindable names for object-level value identifiers and then declare a datatype $expr$ for the above ML expressions:

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

In this declaration, types tell the system which data constructors are binders and how their arguments are bound. For example in `let fun f x = e1 in e2 end` there is a binding occurrence of `f` whose scope is both `e1` and `e2`; and another of `x` whose scope is just `e1`. These binding scopes are reflected by the argument type of `Letf` in the declaration of `expr`. This declarative specification of binding structure is particularly useful because FreshML endows datatypes like `expr` with two crucial properties:

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

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

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

Indeed, FreshML provides automatic language-level support for the common informal idiom which refers to α -equivalence classes 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` of type `expr → name → expr → expr` suffices for `subst e1 x e2` to compute (a representation of) the ML expression obtained by capture-avoiding substitution of the expression represented by `e1` for all free occurrences of the value identifier named `x` in the expression represented by `e2`.

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

Note how simple `subst` is!—only the first clause of the declaration is not ‘boiler plate’ [15]. In particular the clauses dealing with substitution under a binder only have to specify the result when the bound name is sufficiently fresh. Section 2 shows how FreshML ensures only this case arises during evaluation; and `subst` is a total function because by Sect. 5 values of type `expr` represent α -equivalence classes of expressions in the ML fragment.

The Abstractness property says FreshML is *correct* as a language for ‘programming modulo α -equivalence’, and the Concreteness property says it is *expressive*. The two properties oppose each other and it is tricky to make them co-exist. We were guided to the design in [25] that achieves this by using the ‘FM’ mathematical model of binding in terms of name-swapping [10, 13, 24]. This model has a notion of an object’s *support* which specialises to the set of free names when it is an α -equivalence class of parse trees involving binders. We put forward the following informal thesis relating what is sometimes called the ‘Barendregt variable convention’ [3, page 26] to this notion of support.

Thesis: *when people carry out constructions on α -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 in [25] enforces the condition ‘freshly chosen names do not occur in the support of the final result’ 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. The result is a *pure* functional programming language; static freshness inference ensures the dynamics of replacing insufficiently fresh names on the fly is referentially transparent. Purity makes reasoning about program properties simpler and is desirable, but our static freshness inference which achieved it had a drawback: since freshness is only an approximation of ‘not-in-the-support-of’, the type-checker inevitably rejects some algorithms that do in fact conform to the above Thesis. Experience showed this happened too often (we discuss why in Sect. 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 the operational semantics of [25] is modified to make the declaration of fresh bindable names *generative* like some other sorts of names (references, exceptions and type names) are in ML. This fact is not at all obvious and requires proof, which we give.

Summary

Section 2 introduces a new version of FreshML with a slightly more ‘effectful’ dynamics than in [25], but considerably simpler type system. Since we wish to promote it, this new version of the language is simply called *FreshML*; the older, more complicated design from [25] will be referred to as *FreshML 2000*. As explained in Sect. 3, the essence of the dynamics is the combination of generative names with *name-swapping*. To prove that the FreshML type system controls the dynamics sufficiently for object-level α -equivalence to coincide with meta-level operational equivalence (the Abstractness property), in Sect. 4 we describe a denotational semantics of FreshML in the *FM-sets* model of binding [13] that gave rise to the design of FreshML 2000 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 whose details are omitted in this extended abstract) 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] for FreshML 2000. Section 7 describes our experiences implementing FreshML. From the programmer’s point of view, we claim that FreshML extends ML with elegant and practical constructs for declaring and manipulating syntactical data involving binding operations. We give a few examples: capture-avoiding substitution, α -equivalence, computation of free variables (Fig. 5), and normalisation by evaluation (Fig. 7); more 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 introduces the new version of FreshML. Compared with [25], we do without freshness inference in the type system and use a generative operational semantics for fresh names. To make the presentation more accessible, here we use a cut-down language, which we call *FreshML-Lite*, combining the principal novelties of FreshML with a pure functional subset of ML.

Types $\tau \in \text{Ty}$	$::=$	Values $v \in \text{Val}$	$::=$
bindable name	name	atom	a
abstraction type	$\langle \text{name} \rangle \tau$	abstraction	$\langle a \rangle v$
unit type	unit	pair	(v, v)
product type	$\tau \times \tau$	unit	$()$
function type	$\tau \rightarrow \tau$	closure	$[E, x(x) = e]$
data type	δ	constructed	C
			$C v$
Patterns pat	$::=$		
variable	x		
abstraction	$\langle x \rangle x$		
pair	(x, x)		
Declarations dec	$::=$		
value	val $pat = e$		
fresh atom	fresh x		
recursive function	fun $x(x) = e$		
sequential	$dec\ dec$		
Expressions e	$::=$		
value identifier	x		
constructor	C		
atom equality test	$e = e$		
atom swapping	swap e, e in e		
atom abstraction	$\langle e \rangle e$		
unit	$()$		
pair	(e, e)		
application	$e e$		
case split	case e of $(C\ x \Rightarrow e \mid \dots)$		
local declaration	let dec in e end		
Environments	Typing contexts	States	
$E \in \text{VId} \xrightarrow{\text{fin}} \text{Val}$	$\Gamma \in \text{VId} \xrightarrow{\text{fin}} \text{Ty}$	$\bar{a} \in \mathbb{P}_{\text{fin}} \mathbb{A}$	
Top-level datatype declaration			
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*¹ and which are the elements of a fixed, countably infinite set \mathbb{A} . The type $\langle \text{name} \rangle \tau$ of *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 τ . As we see below, FreshML-Lite’s semantics (both operational and denotational) identifies 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 abstraction type are introduced with expressions of the form $\langle e_1 \rangle e_2$ and eliminated via a local declaration with an abstraction pattern, **let val** $\langle x \rangle y = e$ in e' **end**.

FreshML-Lite’s type system is quite standard compared with the one given in [25] (the latter infers freshness information in addition to conventional ML typing properties). Figure 2 gives a selection of

¹The terminology 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

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

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

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

Values

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

$$\frac{a \in \mathbb{A} \quad \vdash v : \tau}{\vdash \langle a \rangle v : \langle \text{name} \rangle \tau} \quad (10)$$

$$\frac{\vdash v_1 : \tau_1 \quad \vdash v_2 : \tau_2}{\vdash (v_1, v_2) : \tau_1 \times \tau_2} \quad (11)$$

Figure 2. FreshML-Lite typing (excerpt)

the typing rules. Comparing rules (3), (6) and (10) with rules (4), (7) and (11) 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 3.1). However, the two types have different dynamic properties. We have chosen to remain close to the Definition of Standard ML [18] and specify the dynamics using inductively defined evaluation relations of the form

$$\bar{a}, E \vdash e \Downarrow v, \bar{a}' \quad (21)$$

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

where the *states* \bar{a} and \bar{a}' are finite subsets of \mathbb{A} and the *value environments* E and E' are finite functions mapping value identifiers to values.² Figure 3 gives a selection of the evaluation rules. Rules (14) and (18) 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.

²In contrast to FreshML 2000 where the environment-style operational semantics is necessary for type soundness, it is quite possible to use a ‘substituted-in’ formulation of FreshML’s dynamics, in which values form a subset of expressions.

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow a, \bar{a}''}{\bar{a}, E \vdash e_1 = e_2 \Downarrow \text{true}, \bar{a}''} \quad (12)$$

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

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a_1, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow a_2, \bar{a}'' \quad \bar{a}'', E \vdash e_3 \Downarrow v, \bar{a}'''}{\bar{a}, E \vdash \text{swap } e_1, e_2 \text{ in } e_3 \Downarrow (a_1 a_2) \cdot v, \bar{a}'''} \quad (14)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow a, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow v, \bar{a}''}{\bar{a}, E \vdash \langle e_1 \rangle e_2 \Downarrow \langle a \rangle v, \bar{a}''} \quad (15)$$

$$\frac{\bar{a}, E \vdash e_1 \Downarrow v_1, \bar{a}' \quad \bar{a}', E \vdash e_2 \Downarrow v_2, \bar{a}''}{\bar{a}, E \vdash (e_1, e_2) \Downarrow (v_1, v_2), \bar{a}''} \quad (16)$$

$$\frac{\bar{a}, E \vdash \text{dec} \Downarrow E', \bar{a}' \quad \bar{a}', E + E' \vdash e \Downarrow v, \bar{a}''}{\bar{a}, E \vdash \text{let } \text{dec} \text{ in } e \text{ end} \Downarrow v, \bar{a}''} \quad (17)$$

$$\frac{\bar{a}, E \vdash e \Downarrow \langle a \rangle v, \bar{a}' \quad a' \in \mathbb{A} - \bar{a}' \quad v' = (a a') \cdot v}{\bar{a}, E \vdash \text{val } \langle x_1 \rangle x_2 = e \Downarrow \{x_1 \mapsto a', x_2 \mapsto v'\}, \bar{a}' \cup \{a'\}} \quad (18)$$

$$\frac{\bar{a}, E \vdash e \Downarrow (v_1, v_2), \bar{a}'}{\bar{a}, E \vdash \text{val } (x_1, x_2) = e \Downarrow \{x_1 \mapsto v_1, x_2 \mapsto v_2\}, \bar{a}'} \quad (19)$$

$$\frac{a \in \mathbb{A} - \bar{a}}{\bar{a}, E \vdash \text{fresh } x \Downarrow \{x \mapsto a\}, \bar{a} \cup \{a\}} \quad (20)$$

Figure 3. FreshML-Lite evaluation (excerpt)

The state may grow during evaluation: if (21) holds, Lemma 2.4 below shows that $\bar{a} \subseteq \bar{a}'$ (and similarly for (22)). The features that cause it to grow are not only declarations of fresh atoms (see rule (20)), but also value declarations involving abstraction patterns: see rule (18). This rule lies at the heart of our treatment of binders. It says that when matching an abstraction pattern $\langle x \rangle y$ against an 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 relations (21) and (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 (18) 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.

FreshML-Lite permits explicit name-swapping in expressions with the syntax $\text{swap } e_1, e_2 \text{ in } e_3$; typing and evaluation rules are (2) and (14) in Figs 2 and 3. Name-swapping can express the operation $e @ x$ from [25] of *concreting* an abstraction at a (fresh) atom, using

`let val $\langle x' \rangle y = e$ in swap x, x' in y .`

Here is a more typical example, using swap -expressions to replace the bound names in a number of abstractions with the same name.

Example 2.1. If $eq : \tau \times \tau \rightarrow \text{bool}$ gives a notion of equality for τ -values, then the following function eqa of type $\langle \text{name} \rangle \tau \times \langle \text{name} \rangle \tau \rightarrow \text{bool}$ correctly computes ‘equality modulo α -conversion’ for $\langle \text{name} \rangle \tau$ -values:

```
fun eqa (p) =
  let val (z, z') = p
      val <x>y = z
      val <x'>y' = z'
  in
    eq (swap x, x' in y, y')
  end.
```

Compare this with the characterisation of α -equivalence via swapping of [32, Theorem 1]. \square

Remark 2.2. The full version of FreshML extends FreshML-Lite with a number of useful features, as well as including standard features from ML such as polymorphism, references and exceptions.

- Compound patterns are allowed, in which abstraction patterns can be nested with other forms of pattern. For example, the declaration of eqa from Example 2.1 can be simplified to:

```
fun eqa (<x>y, <x'>y') = eq (swap x, x' in y, y').
```

- Rather than having just one type name of bindable names, users can declare as many distinct such types as they need; they form a subclass of the equality types and expressions can be polymorphic in the type of bindable names used.
- Abstraction types $\langle bty \rangle ty$, with bty a declared type of bindable names, are a special case of abstraction types $\langle ety \rangle ty$ where ety is an arbitrary equality type. Their values are written $\langle\langle v_1 \rangle\rangle v_2$ (so $\langle a \rangle v$ is a synonym for $\langle\langle a \rangle\rangle v$ when a is an atom). The semantics ensures that these values are indistinguishable up to renaming all the free atoms in the value v_1 (of equality type ety) occurring in the value v_2 (of type ty); this permits simple representations of binding constructs in which the number of bound names is not fixed, such as the ML matching constructs (which bind all the variables occurring in a pattern), or Scheme’s general form of let -expression with a list of declarations.
- If ety and ety' are equality types, then so is $\langle ety \rangle ety'$ and the built-in equality function at that type is the appropriate form of α -equivalence (cf. Example 2.1).

We conclude this section by giving some ‘sanity checks’ on FreshML-Lite’s typing and evaluation rules.

Lemma 2.3. Given value and typing environments E and Γ , 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 2.4. 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 π between $\bar{a}' - \bar{a}$ and $\bar{a}'' - \bar{a}$ such that v' is obtained from v by applying π to the atoms occurring in it. \square

3 The essence of FreshML is swapping

Although in rule (18) of Fig. 3 one can replace swapping $(a a') \cdot v$ with renaming $[a := a'] \cdot v$, because a' does not occur in v , nevertheless swapping rather than renaming underlies our treatment of fresh bindable names. This is because it is simpler to use a totally defined operation that can be applied whether or not a' occurs in v ; but if a'

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

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

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

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

etc.

Figure 4. Translating FreshML-Lite into ML+swap

does occur in v , renaming can have bad properties, whereas swapping is well-behaved. For example if a, a' and a'' are distinct, $\langle a' \rangle a$ and $\langle a'' \rangle a$ are equivalent, but $[a := a'] \cdot (-)$ sends the first to $\langle a' \rangle a'$ and the second to the inequivalent value $\langle a'' \rangle a'$. We might repair this familiar problem of ‘capture’ of free names by binders with a theory of capture-avoiding renaming, but a much simpler solution is just to use swapping $(a a') \cdot (-)$; its self-inverse nature has excellent properties which include preserving α -equivalence. Indeed, a growing body of evidence shows that name-swapping, and more generally permutations of names, are very useful for describing properties of syntax involving binders: see [5, 6, 11, 24, 32].

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

Theorem 3.1. The above translation preserves and reflects FreshML-Lite typing and evaluation: $\Gamma \vdash e : \tau$ 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 it is ‘computationally adequate’: if the translations of two phrases are contextually equivalent in ML+swap, then they are already contextually equivalent in FreshML-Lite. (See Definition 5.3 for a precise definition of contextual equivalence.) The converse is not true; the translation is far from being ‘fully abstract’. This is because abstraction values in FreshML-Lite are translated to (atom,value)-pairs in ML+swap and the identity of the first component can be discovered there in a way that we shall prove in Sect. 5 is impossible in FreshML-Lite. For example, the results in Sect. 5 show 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 ( $\langle x \rangle x, \langle y \rangle y$ ) end,
let fresh x in ( $\langle x \rangle x, \langle x \rangle x$ ) end.
```

Under the translation they become the contextually inequivalent expressions

```
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 distinguished in ML+swap by the context

```
let val x = [-] in #1(#1(x)) = #1(#2(x)) end
```

(where #1 and #2 are ML notation for first and second projection functions). It might seem then that we could better mimic $\langle \text{name} \rangle \tau$ in ML+swap with an abstract type with underlying representation $\text{name} \times \tau$. However, being abstract, we would lose the Concreteness property mentioned in the Introduction, i.e. the ability to match against patterns involving abstraction, which seems so convenient in practice (see also Remark 5.7).

4 Denotational semantics

In this section we give meaning to FreshML-Lite types and expressions with a denotational semantics in the universe of *FM-sets*. This permutation model of set theory devised by Fraenkel and Mostowski in the 1930s is shown in [13] to provide a syntax-independent mathematical model of fresh bindable names and α -conversion, by expressing those concepts purely in terms of swapping names. This mathematics gave rise to the FreshML 2000 design [25]. We saw in the previous section that the dynamics of FreshML’s treatment of bindable names reduces to two things: a dynamically generated supply of 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 α -equivalence.

What is an FM-set? Ordinary sets are members of a cumulative hierarchy obtained by starting with nothing (\emptyset) and continuing transfinitely, at each stage forming new sets by taking powersets. The universe of FM-sets differs just in that 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), and at each stage take only the set of subsets that possess a *finite support*. By definition a set \bar{a} of atoms is a support for X if for all $a, b \in \bar{a}$, X equals $(a b) \cdot X$. Here $(a b) \cdot X$ denotes the set obtained from X by swapping a and b wherever they appear in it (hereditarily). It is the case (though not obviously so: see [13, Proposition 3.4]) that every member X of the universe of FM-sets possesses a *smallest* finite support, written $\text{supp}(X)$.

The only restriction we have when doing FM-set theory is that we must remain within the universe of sets with finite support. The axiomatic development of [10] shows that nearly all logical and set-theoretical constructs have this property. Just axioms of choice, the ability to form a set consisting of an *arbitrary choice* of infinitely many objects of the universe, are limited. 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 an FM-set.

FreshML-Lite features fixpoint recursion in both types and expressions. It is well-known how to use domain theory to give denotational semantics for this. Here we use a domain theory in FM-sets; this gives us access to the FM-set former for atom-abstractions [13, Sect. 5] to give meaning to FreshML-Lite’s abstraction types $\langle \text{name} \rangle \tau$. We only need a relatively simple notion of domain, namely ω -complete partial orders (cpo), within the FM-sets universe:

Definition 4.1. An FM-cpo D is an FM-set equipped with a finitely supported partial order $\sqsubseteq_D \subseteq D \times D$ (often written just \sqsubseteq) such that any finitely supported ω -chain $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots \in D$ (i.e. one for which there is a single finite set of atoms \bar{a} with $\text{supp}(d_n) \subseteq \bar{a}$ for all n) has a least upper bound (lub). We also assume $\text{supp}(D) = \text{supp}(\sqsubseteq_D) = \emptyset$ (i.e. D and \sqsubseteq_D do not depend on particular atoms). A morphism of FM-cpos $f : D \rightarrow D'$ is a function $D \rightarrow D'$ that preserves \sqsubseteq , swapping (i.e. $f((a b) \cdot d) = (a b) \cdot f(d)$), and lubs of finitely supported ω -chains. \square

Such a D need not have lubs of arbitrary ω -chains. For example the denotation of `name` \rightarrow `unit` turns out to be isomorphic to the FM-cpo of subsets of \mathbb{A} that are either finite or whose complement is finite, partially ordered by inclusion; if we enumerate the elements of $\mathbb{A} = \{a_0, a_1, a_2, \dots\}$,³ 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 lub in this FM-cpo.

Our semantics is *monadic* in the sense of Moggi [19]: each type τ is assigned an FM-cpo $\llbracket \tau \rrbracket$ whose elements are used to give meaning to FreshML-Lite values, $\vdash v : \tau$; whereas (closed) expressions of type τ are assigned elements of $\mathbb{T}[\llbracket \tau \rrbracket]$, where \mathbb{T} is a particular ‘dynamic allocation’ monad on FM-cpos. We define \mathbb{T} below, but first here is the definition of $\llbracket \tau \rrbracket$ as τ ranges over FreshML-Lite types.

- $\llbracket \text{unit} \rrbracket \stackrel{\text{def}}{=} \{1\}$, $\llbracket \text{name} \rrbracket \stackrel{\text{def}}{=} \mathbb{A}$: Each FM-set X determines a discrete FM-cpo taking \sqsubseteq as equality on X . The one-element set interprets `unit` and the set of atoms \mathbb{A} interprets `name`.
- $\llbracket \delta_n \rrbracket \stackrel{\text{def}}{=} D_n$: For datatypes we use minimal solutions to simultaneous recursive domain equations corresponding to the datatype declaration. We can construct them by standard techniques using colimits of embedding-projection pairs (see [1, Sect. 5] for example), which transfer smoothly to FM-cpos; we omit 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 , much like the atom-abstraction FM-set in [13, 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 $a_1 = a_2$ and $d_1 \sqsubseteq_D d_2$, or $a_2 \notin \text{supp}(d_1)$ and $(a_1 a_2) \cdot d_1 \sqsubseteq_D d_2$ (cf. Example 2.1). Overloading the notation, we write $[a]d$ for the element of $[\mathbb{A}]D$ given by the equivalence class of the pair (a, d) . We 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, given as for ordinary cpos by ordered pairs with componentwise ordering.
- $\llbracket \tau \rightarrow \tau' \rrbracket \stackrel{\text{def}}{=} \llbracket \tau \rrbracket \rightarrow \mathbb{T}[\llbracket \tau' \rrbracket]$: The function FM-cpo $D \rightarrow D'$ of D and D' consists of all finitely supported subsets of $D \times D'$ that

³Any such bijection of the set of atoms with the natural numbers is external to the FM-sets universe since it cannot be finitely supported.

are total, monotone functions from D to D' preserving least upper bounds of finitely supported chains. To interpret function types we combine this construct with the dynamic allocation monad \mathbb{T} described below. 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 $\mathbb{T}[\llbracket \tau' \rrbracket]$).

\mathbb{T} is the analogue for the category of FM-cpos of one of the dynamic allocation monads on a presheaf category used by Stark [30] to model the Pitts-Stark ν -calculus [26]. Each FM-cpo TD can be constructed as the quotient of $(\mathbb{P}_{\text{fin}} \mathbb{A} \times D)_{\perp}$ by a suitable equivalence relation; we omit the details here and just note that TD carries the following structure:

- A (mono)morphism $\eta : D \rightarrow TD$ (the monad unit).
- A least element $\perp \in TD$.
- A morphism $\nu : [\mathbb{A}]D \rightarrow D$ (‘restriction’) satisfying for all $a, b \in \mathbb{A}$ and $x \in TD$ that

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

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

TD is the minimal cpo with these properties, i.e. it has a certain category-theoretic universal property, which we omit; and we can use that to define the lifting part of the monad structure and verify the usual monad laws. We can present each non-bottom element of TD 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). The element (26) models a convergent FreshML-Lite expression whose evaluation creates some fresh atoms \bar{a} and returns a value denoted by d . We need a dynamic allocation monad satisfying (24) and (25), rather than a simpler one just pairing up name sets and values, to prove Theorem 5.6.

We can now construct the denotations of FreshML-Lite values, expressions and declarations:

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

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

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

When we refer to ‘some/any atom’ above, we are really using the *freshness quantifier* \mathbb{V} of [13]: we choose some atom satisfying the condition, but in fact any such one will do.

5 Correctness

In this section we show the denotational semantics of Sect. 4 matches the operational semantics closely enough that we can prove that values of recursively defined FreshML-Lite datatypes represent α -equivalence classes of object-level syntax (the Abstraction property from the Introduction). For simplicity we take as object-language the familiar untyped λ -calculus, but the results easily generalise to other algebraic signatures with binders. We noted in the previous section that the denotation $\llbracket \delta \rrbracket$ of a declared FreshML-Lite datatype δ is a recursively defined FM-cpo. When δ arises from an algebraic signature with binders, it only involves product- and abstraction-, but not function-types; in this case $\llbracket \delta \rrbracket$ is a discrete FM-cpo given by an *inductively-defined FM-set*.

Example 5.1. Given the datatype declaration

```
datatype lam = Var of name
           | Lam of (name) lam
           | App of lam × lam
```

then $\llbracket \text{lam} \rrbracket$ is isomorphic to the discrete FM-cpo given by the inductively defined FM-set $\mu X. (\mathbb{A} + [\mathbb{A}]X + X \times X)$. Previous work [13, Theorem 6.2] shows this to be in bijection with the set $\Lambda(\mathbb{A})/\equiv_\alpha$ of α -equivalence classes $[t]_{\equiv_\alpha}$ of untyped λ -terms t with variables in \mathbb{A} :

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

(Swapping is given by $(a b) \cdot [t]_{\equiv_\alpha} = [(a b) \cdot t]_{\equiv_\alpha}$, where $(a b)$ interchanges all instances of a and b in t ; the support of $[t]_{\equiv_\alpha}$ is 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 λ -terms $t \in \Lambda(\mathbb{A})$ and values $(t)_v$ of type 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 $\Lambda(\mathbb{A})/\equiv_\alpha \cong \llbracket \text{lam} \rrbracket$ above, $[t]_{\equiv_\alpha}$ is identified with $\llbracket \vdash (t)_v : \text{lam} \rrbracket$. Therefore for all t and t' ,

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

□

Thus denotations of values of FreshML-Lite datatypes like lam are in bijection with α -equivalence classes of terms of the object-language they represent. To connect this to operational behaviour of FreshML-Lite programs we have to do two things. First, we examine the translation of object-level terms into expressions rather than values, since these are what the programmer writes, and second, we relate equality of denotation of expressions to an appropriate notion of operational behaviour (Definition 5.3).

To tackle the first issue, still using an object-language of λ -terms, note that atoms \mathbb{A} and value identifiers $\mathbb{V}\text{Id}$ are both countably infinite sets. By enumerating each, fix some explicit bijection $a_i \leftrightarrow x_i$, then translate the λ -terms t of Example 5.1 into FreshML-Lite expressions $(t)_e$ as follows:

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

Lemma 5.2. One can show by induction on the structure of t that if its free variables are among $\{a_1, \dots, a_n\}$ and if Γ is the typing context mapping the corresponding value identifiers x_1, \dots, x_n to name , then $\Gamma \vdash (t)_e : \text{lam}$ holds and for any $\rho \in \llbracket \Gamma \rrbracket = \mathbb{A}^n$

$$\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \nu\emptyset \cdot \llbracket \vdash (\rho t)_v : \text{lam} \rrbracket \in \mathbb{T}[\llbracket \text{lam} \rrbracket] \quad (28)$$

where ρt is the λ -term obtained by simultaneous capture-avoiding substitution of $\rho(x_i)$ for a_i in t (for $i = 1, \dots, n$).

PROOF. When it comes to the step for λ -abstractions in the proof of (28), 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. □

We now know for all t and t' that

$$t \equiv_\alpha t' \quad \text{iff} \quad \llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket \quad (29)$$

where Γ is as in Lemma 5.2. For if $t \equiv_\alpha t'$, then $\rho t \equiv_\alpha \rho t'$ for any ρ ; so from (27) we get $\llbracket \vdash (\rho t)_v : \text{lam} \rrbracket = \llbracket \vdash (\rho t')_v : \text{lam} \rrbracket$ and therefore $\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket(\rho)$ by (28); since this holds for any ρ , we have the left-to-right implication in (29). Conversely, if $\llbracket \Gamma \vdash (t)_e : \text{lam} \rrbracket(\rho) = \llbracket \Gamma \vdash (t')_e : \text{lam} \rrbracket(\rho)$ holds for any ρ , from (28) again 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 ρ 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 $t \equiv_\alpha t'$. □

Turning to the second issue mentioned above, namely relating equality of denotation of expressions to 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 observable behaviour. We take programs to be closed expressions of type unit , and their observable behaviour to be whether they evaluate, ignoring any fresh atoms created along the way. The following more precise definition uses typing and evaluation as defined in Sect. 2, and the notion of a *context* $\mathcal{C}[-]$. As usual, contexts are generated by the grammar in Fig. 1 augmented by a placeholder ‘ $-$ ’. $\mathcal{C}[e]$ then denotes the result of 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} contains the atoms of E . Let ρ be given by $\rho(x) = \llbracket E(x) : \Gamma(x) \rrbracket$, for $x \in \text{dom}(\Gamma)$.

- (a) If $\bar{a}, E \vdash e \Downarrow v, \bar{a}'$, then $\llbracket \Gamma \vdash e : \tau \rrbracket(\rho)$ 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. (a) is proved 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 based on type-indexed *logical relations* relating domain elements to FreshML-Lite values and expressions. The construction is complicated because, as for ML, FreshML allows function types in recursively defined datatypes (see Fig. 7 in Sect. 6 for an example), precluding a simple inductive technique; instead, we deduce the existence of the logical relations using the general theory of *minimal invariant relations* [23] applied in the setting of FM-cpos (details omitted in this extended abstract). \square

Parts (a) and (b) together 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 τ is an equality type. \square

Theorem 5.6. (Correctness of representation) Under the translation of λ -terms t into FreshML-Lite expressions $(t)_e$ of type lam given above, α -equivalence of λ -terms corresponds to contextual equivalence of FreshML-Lite expressions: given two λ -terms t and t' , with free variables among $\{a_1, \dots, a_n\}$ say, letting Γ 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 (29) to get the desired conclusion. \square

We emphasise that although this correctness theorem is for λ -terms, similar results hold for object languages specified by a general notion of binding signature (such as the nominal signatures of [32, Definition 1], or the nominal algebras of [14]) versus datatypes in FreshML-Lite whose declarations are derived from the signature in the simple declarative fashion discussed in the Introduction.

Remark 5.7. 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 to manipulate these representations. FreshML's abstraction patterns are very useful in this respect. In particular we can use them to provide an important facility, namely *the ability to recognise that a meta-level expression represents some object-level term*. For an impure functional programming language this cannot just be a matter

```

1 (* remove : name -> name list -> name list *)
2 fun remove x [] = []
3   | remove x (y:::ys) =
4     if x = y then remove x ys
5     else y:::(remove x ys);
6 (* fv : lam -> name list *)
7 fun fv (Var x) = [x]
8   | fv (Lam(<x>t)) = remove x (fv t)
9   | fv (App(t1,t2)) = (fv t1) @ (fv t2);
10 (* is_closed t : lam -> bool *)
11 fun is_closed t = ((fv t) = [])

```

Figure 5. Testing for closed λ -terms in FreshML

of typing and termination; because of side-effects from references and exceptions, the full FreshML has terminating closed expressions of type lam from Example 5.1 which are not in the image of the translation $t \mapsto (t)_e$ of λ -terms. However, even the cut-down FreshML-Lite of Sect. 2 has this property. For example

$$\text{genvar} \stackrel{\text{def}}{=} \text{let fresh } x \text{ in } (\text{Var } x) \text{ end} \quad (30)$$

evaluates to ‘some fresh object-level variable’. Once we identify $\llbracket \text{lam} \rrbracket$ with $\Lambda(\mathbb{A})/\equiv_{\alpha}$ as in Example 5.1, the denotational semantics of genvar is $\llbracket \emptyset \vdash \text{genvar} : \text{lam} \rrbracket = \nu\{a\} \cdot ([a]_{\equiv_{\alpha}}) \in \mathbb{T}[\llbracket \text{lam} \rrbracket]$ (and $[a]_{\equiv_{\alpha}}$ is just $\{a\}$); from this and Sect. 5 it follows that genvar is not contextually equivalent to $(t)_e$ for any closed λ -term t . Nevertheless, we can easily write FreshML-Lite boolean-valued functions to recognise an expression as the encoding of some object-level term; for example is_closed declared in Fig. 5 (for legibility we used syntax for nested patterns and standard list constructs). This uses helper functions remove for removing an atom from a list of atoms and fv for computing the free variables of a λ -term (possibly with repeats); and then is_closed tests whether that list is empty. For example is_closed genvar evaluates to false (creating a fresh atom as it does so). Our results can 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 $\emptyset \vdash \text{is_closed } e \approx_{\text{ctx}} \text{true} : \text{bool}$ if and only if $\emptyset \vdash e \approx_{\text{ctx}} (t)_e : \text{lam}$ holds for some closed λ -term t .

6 Freshness inference

Recall from Sect. 4 the important FM-sets notion of *finite support* (and the notation $\text{supp}(X)$ for the smallest set of atoms supporting an FM-set X). In informal reasoning about α -equivalence classes by choosing representatives with sufficiently fresh bound names (c.f. the ‘Barendregt variable convention’ [3, page 26]), it seems that the end result is always 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 (for if $a, b \notin \text{supp}(X)$, then swapping a and b has no effect on X , i.e. $(a \ b) \cdot X = X$). This is the informal Thesis put forward in the Introduction. (See [11, Sect. 2.3] for some mathematical justification of it.) The definition of the denotational semantics at the end of Sect. 4 contains several examples of this phenomenon, for example.

FreshML 2000 [25] enforces that freshly chosen names not be in the support of final results at compile time: the type-checker builds up additional information about a relation of *freshness* between value identifiers and expressions, $x \# e$, which is a sound decidable approximation to the ‘not-in-the-support-of’ relation $a \notin \text{supp}(\llbracket e \rrbracket)$ in the denotational semantics. (Semantic ‘not-in-the-support-of’ is undecidable for the usual recursion-theoretic reasons.) This allows FreshML 2000 to reject phrases causing observable side ef-


```

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

```

Figure 6. Free variables of λ -terms in FreshML 2000

facts when fresh names are allocated dynamically. For example, x is most definitely not fresh for the expression $\text{Var } x$ (corresponding to the fact that $a \in \text{supp}(a) = \{a\}$); and FreshML 2000 will not admit *genvar* declared in (30) as a well-typed expression of type lam . In fact FreshML 2000’s static freshness inference allows one to simplify its dynamics by not threading through evaluation a state containing the names generated so far. Not only does a correctness result like Theorem 5.6 hold for FreshML 2000, but it also satisfies ‘no junk’ properties; for example, every closed terminating expression of type lam in the fragment of FreshML 2000 without references and exceptions is of the form $(t)_e$ for some closed λ -term t . As we saw in Sect. 5, FreshML-Lite has some ‘junk’, but at least it is possible to recognise it dynamically (Remark 5.7).

So static freshness-checking gives a more effect-free functional language with better programming laws. However our experiences implementing and using freshness inference indicate that the price is too high, as we shall discuss.

As far as implementation goes, unfortunately the freshness inference algorithm has to proceed not solely on the structure of expressions, but also sometimes on the structure of (inferred) types. This is due to the notion of *purity*—judgements which state that values of *certain types* (such as `bool`, `int`, or `string`, for example) can never contain atoms in their support. We need this to treat many seemingly innocuous uses of freshened bound names. For example to deduce that $\text{fn } \langle x \rangle y \Rightarrow y+1$ respects the conditions imposed by freshness checking, we must deduce $x \# y+1$. To do that we examine the type of the body of the match, which is `int`, and with purity judgements can observe that a value of type `int` cannot ever contain any atoms in its support; so we deduce $x \# y+1$. More complicated problems of this kind arise in recursive datatype declarations; to deduce which kinds⁴ 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 as in the procedure for maximising equality in ML [18, Sect. 4.9]. With purity, more programs are typeable, but type-checking gets harder for the user to understand and predict.

Experience writing syntax-manipulating programs in FreshML 2000 was one of the main motivations for developing a simpler FreshML without freshness inference. We were too frequently forced to adopt an obtuse coding style to get programs to pass the freshness checks. For example, the code in Fig. 5 does not type-check in FreshML 2000, because at line 8 the freshness checking algorithm cannot deduce that $x \# (\text{remove } x \text{ } fv \ t)$ holds. In fact when evaluating an application of the fv function by matching against the clause at line 6, the final result does not depend upon the particular fresh atom associated with the bound name x , because an atom is not in the support of a list of atoms that has had that atom removed. This fact has a simple proof by induction

⁴Both FreshML 2000 and FreshML permit the declaration of different types of bindable names, whose values range over disjoint copies of \mathbf{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 (unit->sem) -> sem (* function *)
9   | N of neu (* neutral *)
10 and neu =
11   V of name (* variable *)
12   | A of neu*sem; (* neutral appn *)
13 (* reification reify : sem -> lam *)
14 fun reify(L f) =
15   let fresh x:name in
16     Lam(<x>(reify(f(fn () => N(V x)))))) end
17   | reify(N n) = reifyn n
18 and reifyn(V x) = Var x
19   | reifyn(A(n,d)) =
20     App(reifyn n, reify d);
21 (* evals : (unit->sem)list -> lam -> sem *)
22 fun evals [] (Var x) = N(V x)
23   | evals((x,v)::env)(Var y) =
24     if x = y then v()
25     else evals env (Var y)
26   | evals env (Lam(<x>t)) =
27     L(fn v => evals((x,v)::env) t)
28   | evals env (App(t1,t2)) =
29     (case evals env t1 of
30       L f => f(fn () => evals env t2)
31       | N n => N(A(n,evals env t2)));
32 (* evaluation eval : lam -> sem *)
33 fun eval t = evals [] t;
34 (* normalisation norm : lam -> lam *)
35 fun norm t = reify(eval t)

```

Figure 7. Normalisation by evaluation in FreshML

on the length of the list, but this is not something our freshness-checking algorithm is able to use at the point it needs to verify the condition $x \# (\text{remove } x \text{ } fv \ t)$. For this particular example we can circumvent the problem by using a *remove* function of type $\langle \text{name} \rangle (\text{name list}) \rightarrow \text{name list}$, as in Fig. 6.⁵ At line 8 of this figure the previous problem has gone away 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). FreshML 2000 spots this at compile-time and allows this new version of fv as a well-typed expression of type $\text{lam} \rightarrow \text{name list}$.

Figure 7 gives a subtler example of the shortcomings in static freshness inference in FreshML 2000, at the same time showing off how FreshML can express rather clearly non-trivial syntax-manipulating algorithms; in this case an algorithm computing the normal form of an untyped λ -term (if there is one) using *normalisation by evaluation* in a form suggested to us by Thierry Coquand [private communication] and adapted for call-by-value by Olivier Danvy. (See [2, Sect. 3] and the references there for more on normalisation by evaluation in an untyped setting.) In FreshML the function *norm* has type $\text{lam} \rightarrow \text{lam}$ (and does indeed compute normal forms where they exist); in FreshML 2000, *norm* does not type-check because

⁵Atom inequality, written $x \# y$, is needed in this declaration of *remove* to signal to the freshness-checker that it can use the fact that x is fresh for y when checking the first branch of the conditional.

```

datatype 'a am = In of 'a
              | Ab of <name>('a am);
(* monad unit *)
fun return x = In x;
(* monad lifting
   op>>=: 'a am * ('a->'b am) -> 'b am *)
infix >>= ;
fun ((In x) >>= f) = f x
  | ((Ab (<n>y)) >>= f) = Ab (<n>(y >>= f));
(* forcing at pure types, e.g... *)
fun force (In (s:string)) = s
  | force (Ab (<n>y)) = force y

```

Figure 8. An abstraction monad

the helper function *evals* does not. Checking fails at the clause in the definition of *evals* at lines 26–27, where the system cannot deduce that $x \# (\text{fn } v \Rightarrow \text{evals } ((x, v) :: \text{env}) t)$ 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.

Remark 6.1. We can get around these shortcomings of freshness inference within FreshML 2000 by adopting a *monadic* programming style that mimics the use of the dynamic allocation monad T in the denotational semantics of Sect. 4, using an *abstraction monad*, such as the one in Fig. 8. Using this monad to wrap return types in abstractions, freshness inference only needs to use the simple fact $x \# \langle x \rangle e$; plus the fact that abstractions can be discarded at pure types, as in the function *force* in the figure. The efficiency of this style is questionable, and the resulting programs somewhat obfuscated. Nevertheless, it was the realisation that one can adopt this monadic style in FreshML 2000 that led us to the design of the simpler FreshML and its denotational semantics; and using name abstraction to model name generation (such as in Fig. 8) still has its uses in FreshML: see [12].

We advocate use of FreshML, which does not enforce the Introduction’s Thesis (the property of freshly chosen names not being in the support of final results) at compile-time. However, freshness inference may still be useful for other purposes: it could be used in a program logic for verifying properties of FreshML programs; and information about freshness deduced at compile-time may be useful for optimising run-time implementation, the issue we turn to 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 no modules layer. If so desired, FreshML 2000’s freshness inference can be switched on by starting *freshml* with the command-line argument `--pure`. The web site contains examples of programming with binders in FreshML, including programs calculating the possible labelled transitions from a π -calculus [17] process, using various forms of encoding [12]; and Barthe’s classification algorithm for type-checking injective Pure Type Systems [4]. We invite readers to try FreshML for themselves!

We saw in Sect. 3 that the dynamics of FreshML can be implemented by translating it into ML augmented with a primitive function *swap* : `unit ref` \rightarrow `unit ref` \rightarrow $\alpha \rightarrow \alpha$ for swapping address

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

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

Figure 9. Values with delayed swapping

names in ML values. It is possible to add this to some existing ML implementations using unsafe features.⁶ Since the representation of values in such systems was not designed with swapping in mind, such simple hacks may not yield very efficient implementations of FreshML; but they do enable us to demonstrate FreshML’s novel features for computing with binders integrated into a complete ML system. Shinwell [28] reports on such an experiment with Objective Caml and the resulting *Fresh O’Caml* is available at <http://www.freshml.org/>.

To make atom-swapping more efficient, our implementation of FreshML uses a representation of FreshML values suggested by Mark Shields [private communication] using *delayed swapping*. In this scheme, shown in Fig. 9, values p have at each structural level ‘explicit permutations’ of atoms (represented by finite lists of pairs of atoms, π , 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 (14) from Fig. 3 becomes

$$\frac{\bar{a}, P \vdash e_1 \Downarrow a_1, \bar{a}' \quad \bar{a}', P \vdash e_2 \Downarrow a_2, \bar{a}'' \quad \bar{a}'', P \vdash e_3 \Downarrow c, \bar{a}'''}{c' = \text{cf}((a_1 a_2) :: \square \bullet c)} \quad \bar{a}, P \vdash \text{swap } e_1, e_2 \text{ in } e_3 \Downarrow c', \bar{a}''' \quad (31)$$

where $p \mapsto \text{cf}(p)$ is an auxiliary function converting a non-canonical value to canonical form by pushing the outermost explicit permutation through one structural level and applying it to any atom it meets. This is relatively inexpensive to implement compared with traversing the whole parse tree swapping its atoms.

It seems that in practice most swappings arise from deconstructing abstraction values $\langle a \rangle v$. In the delayed swapping implementation scheme, the evaluation rule for this (cf. rule (18) in Fig. 3) becomes

$$\frac{\bar{a}, P \vdash e \Downarrow \langle a \rangle (\pi \bullet c), \bar{a}' \quad a' \in \mathbb{A} - \bar{a}' \quad p_1 = \square \bullet a' \quad p_2 = (a a') :: \pi \bullet c}{\bar{a}, P \vdash \text{val } \langle x_1 \rangle x_2 = e \Downarrow \{x_1 \mapsto p_1, x_2 \mapsto p_2\}, \bar{a}' \cup \{a'\}} \quad (32)$$

with $(a a')$ appended to π . Another optimisation is to try to avoid choosing a fresh atom a' at all. To make this possible, we can consider ‘garbage collection’ rules that try to reduce the state (set of atoms) by removing atoms not in the support of the results of

⁶Thanks to Claudio Russo for showing how to do this in Moscow ML.

evaluation (properties such as Lemma 2.4 have to be modified if we do this). Then in the above rule it may be the case that $a \in \mathbb{A} - \bar{a}'$ and we can take $a' = a$ and replace $(a a') :: \pi \bullet c$ with $\pi \bullet c$.

We have not yet fully exploited such optimisations. In particular the freshness inference discussed in Sect. 6 could be useful for optimising the dynamics of FreshML. For example, for non-canonical values of *pure* type, when converting to canonical form we can just discard the outermost permutation rather than pushing it down into the value.

8 Related work

Sheard [27, Sect. 13] gives an excellent survey of the problems for metaprogramming caused by statically scoped binders in object terms and some of the solutions that have been proposed. FreshML seems unique among metaprogramming languages in providing language-wide support for object-level α -equivalence while still allowing the user to refer to bound entities by name. Miller [16] proposed incorporating elements of *higher order abstract syntax*, HOAS [22], into an ML-like programming language, ML_λ , with intentional function types $\text{ty} \Rightarrow \text{ty}'$. HOAS also underlies Raffalli's Bindlib Library for O'Caml.⁷ FreshML's underlying theory of binders [13] lifts less 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 [13, Sect. 6]. Also, while ML_λ and Bindlib give no direct access to bound names and their distinctions (name equality and inequality), FreshML does. The price paid for this ability is dynamic generation of fresh names. The seemingly small computational effect (which in fact has rather subtle interactions with higher order functions: see [26]) has a long history in functional programming, from Lisp's *gensym* to more recent advocates, such as [21, 7]. However, the combination with name-swapping and abstraction types $(bty)ty$ is unique to FreshML. Since the original design was published in [25], the use of swapping and freshness to deal with α -equivalence and name restriction has been taken up by others, such as in [6]. Indeed, it was reading this work that inspired us to remove the rather restrictive freshness checking from FreshML 2000's statics and design the new version of FreshML presented here. FreshML 2000's approach to names and binding has also inspired work on open code types in homogeneous metaprogramming languages by Nanevski and Pfenning [20].

FreshML's correctness properties established in Sect. 5 are not obvious, and we had to work quite hard to establish them. We introduced 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 ω -cpos: see [9, 31] for example. The detailed proofs of the results in Sects 4 and 5 are less complicated than the corresponding ones in 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 π -calculus) and spatial locality in general; the work in [5, 11, 12] already takes steps in this direction.

Simplifying FreshML 2000 to produce FreshML as we have described in this paper opens up many interesting possibilities. For one thing, it opens the door to full-scale functional language implementations incorporating our approach to programming with binders, as Shinwell's work on Fresh O'Caml [28] shows. It has also made it possible to support abstraction types for binding data consisting of more than a single atom (see Remark 2.2). Freshness inference is poorly-understood for such types; the main problem is judging which atoms *are* in the support of a value as well as which are not. FreshML neatly removes the need to solve this problem and allows us to provide this generalised form of abstraction.

The features we have described here for programming with binders seem really useful. Perhaps the school of pure, lazy functional programming should have them too—there should be a FreshHaskell! Once again, the simple form of FreshML presented here holds out hope that this might be possible, since it removes the need to get effective static freshness information, which appears to be much harder for a non-strict language than for a strict one. Of course our design is impure—some side-effects of generating fresh names are left exposed in FreshML; in 'FreshHaskell' one would presumably encapsulate them using a monad, mimicking the use of a monad in the denotational semantics of Sect. 4 (cf. the abstraction monad 'a *am* in Fig. 8). The design and implementation 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, Olivier Danvy, Simon Peyton Jones, Claudio Russo, Mark Shields, Keith Wansbrough and the anonymous referees for helpful comments and feedback on this work. Peter White contributed much to the implementation work, which benefitted enormously from use of the O'Caml 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] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report BRICS RS-03-14, BRICS, Department of Computer Science, University of Aarhus, March 2003.
- [3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [4] G. Barthe. Type-checking injective pure type systems. *Journal of Functional Programming*, 9(6):675–698, 1999.
- [5] L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *CONCUR 2002 – Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002. Proceedings*, volume 2421 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, Berlin, 2002.
- [6] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Foundations of Software Science and Com-*

⁷http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/bindlib.html

- putation Structures, 6th International Conference, FOSSACS 2003, Warsaw, Poland. Proceedings, volume 2620 of Lecture Notes in Computer Science, pages 216–232. Springer-Verlag, Berlin, 2003.
- [7] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Advances in Computing Science ASIAN'99, 5th Asian Computing Science Conference*, 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 π -calculus (extended abstract). In *Eleventh Annual Symposium on Logic in Computer Science*, pages 43–54. IEEE Computer Society Press, Washington, 1996.
- [10] M. J. Gabbay. *A Theory of Inductive Definitions with α -Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, University of Cambridge, 2000.
- [11] M. J. Gabbay. The π -calculus in FM. Submitted, September 2002.
- [12] M. J. Gabbay. FM for process calculi that generate fresh names. Submitted, June 2003.
- [13] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
- [14] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *28th International Colloquium on Automata, Languages and Programming, ICALP 2001, Crete, Greece, July 2001. Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer-Verlag, Heidelberg, 2001.
- [15] R. Laemmel and S. L. Peyton Jones. Scrap your boilerplate: A practical approach to generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans*, pages 26–37. ACM Press, 2003.
- [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 *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002, Pittsburgh, Pennsylvania*, pages 206–217. ACM Press, New York, 2002.
- [21] M. Odersky. A functional theory of local names. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 48–59. ACM Press, 1994.
- [22] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 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 the *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (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 *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.
- [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 *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.
- [27] T. Sheard. Accomplishments and research challenges in meta-programming. In *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings.*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.
- [28] M. R. Shinwell. Swapping the atom: Programming with binders in Fresh O'Caml. Submitted, June 2003.
- [29] M. R. Shinwell and A. M. Pitts. *FreshML User Manual*. Cambridge University Computer Laboratory, November 2002. Available at <http://www.freshml.org/docs/>.
- [30] I. D. B. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.
- [31] I. D. B. Stark. A fully abstract domain model for the π -calculus. In *11th Annual Symposium on Logic in Computer Science*, pages 36–42. IEEE Computer Society Press, Washington, 1996.
- [32] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003.