R. Backhouse and J. N. Oliveira (Eds.): MPC 2000, LNCS 1837, pp. 230–255, 2000.

A Metalanguage for Programming with Bound Names Modulo Renaming

Andrew M. Pitts[1] and Murdoch J. Gabbay[2]

[1] Cambridge University Computer Laboratory, Cambridge CB2 3QG, UK
Andrew.Pitts@cl.cam.ac.uk
[2] Department of Pure Mathematics and Mathematical Statistics, Cambridge University, Cambridge CB2 1SB, UK
M.J.Gabbay@dpmms.cam.ac.uk

Abstract. This paper describes work in progress on the design of an ML-style metalanguage FreshML for programming with recursively defined functions on user-defined, concrete data types whose constructors may involve variable binding. Up to operational equivalence, values of such FreshML data types can faithfully encode terms modulo α-conversion for a wide range of object languages in a straightforward fashion. The design of FreshML is 'semantically driven', in that it arises from the model of variable binding in set theory with atoms given by the authors in [7]. The language has a type constructor for abstractions over names (= atoms) and facilities for declaring locally fresh names. Moreover, recursive definitions can use a form of pattern-matching on bound names in abstractions. The crucial point is that the FreshML type system ensures that these features can only be used in well-typed programs in ways that are insensitive to renaming of bound names.

1 Introduction

This paper concerns the design of functional programming languages for *metaprogramming*, by which we mean the activity of creating software systems—interpreters, compilers, proof checkers, proof assistants, and so on—that manipulate syntactical structures. An important part of such activity is the design of data structures to represent the terms of formal languages. The nature of such an *object language* will of course depend upon the particular application. It might be a language for programming, or one for reasoning, for example. But one thing is certain: in all but the most trivial cases, the object language will involve variable binding, with associated notions of free and bound variables, renaming of bound variables, substitution of terms for free variables, and so on. It is this aspect of representing object languages in metaprogramming languages upon which we focus here.

Modern functional programming languages permit user-defined data types, with pattern matching in definitions of functions on these data types.[1] For object

[1] As far as we know, this feature was introduced into functional programming by Rod Burstall: see [2, 1].

languages without variable binding, this reduces the work involved in designing representations to a mere act of declaration: a specification of the abstract syntax of the object language gives rise more or less directly to the declaration of some algebraic data types (mutually recursive ones in general). Consider the familiar example of the language of terms of the untyped lambda calculus

$$t ::= x \mid t\,t \mid \lambda x.t \tag{1}$$

and a corresponding ML data type

```
datatype ltree = Vr of string                    (2)
               | Ap of ltree * ltree
               | Lm of string * ltree
```

where x ranges over some fixed countably infinite set of variable symbols which we have chosen to represent by values of the ML type `string` of character strings. This gives a one-one representation of the abstract syntax trees of all (open or closed) untyped lambda terms as closed ML values of type `ltree`. However, the ML declaration takes no account of the fact that the term former $\lambda x.(-)$ involves variable binding. Thus, if one wishes to identify terms of the object language up to renaming of bound variables (as one often does), such representations are too concrete. It is entirely up to programmers to ensure that their term manipulating programs respect the renaming discipline—an obligation which becomes irksome and error prone for complex object languages, or large programs.

A common way round this problem is to introduce a new version of the object language that eliminates variable binding constructs through the use of de Bruijn indices [4]. For example, 'nameless' lambda terms are given by

$$t' ::= n \mid t'\,t' \mid \lambda\,t' \tag{3}$$

and a corresponding ML data type by

```
datatype ltree' = Vr' of nat                     (4)
                | Ap' of ltree' * ltree'
                | Lm' of ltree'
```

where the indices n are natural numbers, represented by the values of a suitable ML data type `nat`. Closed ML values of type `ltree'` correspond to nameless terms t', which in turn correspond to α-equivalence classes of ordinary lambda terms t (open or closed). Hence functions manipulating lambda terms modulo α-conversion can be defined, and their properties proved, using structural recursion and induction for the algebraic data type `ltree'`. This approach has been adopted for a number of large systems written in ML involving syntax manipulation (such as HOL [8] and Isabelle [14], for example). However, it does have some drawbacks. Firstly, nameless terms are hard for humans to understand and they need translation and display functions relating them to the usual syntax with named bound variables. Secondly, some definitions (such as substitution and weakening the context of free variables) are non-intuitive and error-prone when cast in terms of de Bruijn indices. Lastly, and most importantly, the ML language does not have any built-in support that might alleviate these problems: one usually starts with a specification of an object language in terms of context free grammars and some indication of the binding constructs and has to craft its 'name free' representation by hand. Perhaps more can be done automatically? In this paper we describe an ML-like language with features that address these difficulties and provide improved automatic support for metaprogramming with variable binding constructs. The key innovation is to deduce at compile-time not only traditional type information, but also information about the 'freshness' of object-level variables. This information is used to guarantee that at run-time the observable behaviour of well-typed meta-level expressions is insensitive to renaming bound object-level variables. Thus, users are notified at compile-time if their syntax-manipulating code descends below the level of abstraction which identifies α-equivalent object-level expressions.

Our language design is guided by the mathematical model of binding operations introduced in [7] using a Fraenkel-Mostowski permutation model of sets with atoms. A key feature of this model is that it provides a syntax-independent notion of a name (i.e. an atom) being *fresh* for a given object. For this reason the resulting programming language is called FreshML. Figure 1 gives some sample FreshML declarations which continue the running example of the untyped lambda calculus.[2] They will be used in the rest of this paper to illustrate the features of the new language. We attempt to explain FreshML without assuming knowledge of the mathematics underlying our model of binding; for the interested reader, the intended model of FreshML is sketched in an Appendix to this paper. Sections 2–5 describe the novel features of FreshML compared with ML, namely *atoms*, *freshness*, *atom abstraction/concretion* and pattern matching with *abstraction patterns*. Sections 6–8 discuss the interaction of these features with standard ones for equality, recursive functions and types not involving atoms. It should be stressed that the design of FreshML is still evolving: section 9 discusses some of the possibilities and reviews related work.

Note (Meta-level versus object-level binding). The metalanguage FreshML provides a novel treatment of binding operations in object languages. However, in describing FreshML syntax we treat its various binding constructs in a conventional way—by first giving their abstract syntax and then defining what the free, bound and binding identifiers are in such expressions. This in turn gives rise to a conventional definition of the capture-avoiding substitution $[exp/\mathtt{x}]exp'$ of a FreshML expression exp for all free occurrences of an identifier `x` in an expression exp'. We write $\mathrm{fv}(exp)$ for the finite set of free identifiers in exp.

[2] It will be seen from these declarations that the syntax of function declarations (and case expressions) in FreshML is more like that of CAML [3] than that of Standard ML [12].

```
(* Lambda terms, modulo alpha conversion. *)
datatype lam = Var of atm
             | App of lam * lam
             | Lam of [atm]lam;

(* Encoding of a couple of familiar combinators. *)
val I = new a in Lam a.(Var a) end;
val K = new a in new b in Lam a.(Lam b.(Var a)) end end;

(* A function sub:lam * [atm]lam -> lam
   implementing capture avoiding substitution. *)
fun sub =
 { (t, a.(Var b)) where b=a => t
 | (t, a.(Var b)) where b#a => Var b
 | (t, a.(App(u,v))) => App(sub(t, a.u), sub(t, a.v))
 | (t, a.(Lam b.u)) => Lam b.(sub(t, a.u)) };

(* A function cbv: lam -> lam
   implementing call-by-value evaluation. *)
fun cbv =
 { App(t,u) => case (cbv t) of {Lam e => cbv(sub(cbv u, e))}
 | v => v };

(* A function rem: [atm](atm list) -> (atm list)
   taking a list of atoms with one atom abstracted and removing it. *)
fun rem =
 { a.nil => nil
 | a.(x::xs) where x=a => rem a.xs
 | a.(x::xs) where x#a => x::(rem a.xs) };

(* A function fv: lam -> (atm list)
   which lists the free variables of a lambda term,
   possibly with repeats. *)
fun fv =
 { Var a => a::nil
 | App(t,u) => append(fv t)(fv u)
 | Lam a.t => rem a.(fv t) };

(* Unlike the previous function, the following function, which tries
   to list the bound variables of a lambda term, does not type check
   ---good! *)
fun bv =
 { Var a => nil
 | App(t,u) => append(bv t)(bv u)
 | Lam a.t => a::(bv t) };
```

Fig. 1. Sample FreshML declarations

2 Freshness

Variables of object languages are represented in FreshML by value identifiers of a special built-in type `atm` of *atoms*.[3] Operationally speaking, `atm` behaves somewhat like the ML type `unit ref`, but the way in which dynamically created values of type `atm` (which are drawn from a fixed, countably infinite set $\mathbb{A} = \{a, a', \ldots\}$ of *semantic atoms*) can be used is tightly constrained by the FreshML type system, as described below. Just as addresses of references do not occur explicitly in ML programs, semantic atoms do not occur explicitly in the syntax of FreshML. Rather, they can be referred to via a local declaration of the form

$$\texttt{new a in } \mathit{exp} \texttt{ end} \tag{5}$$

where `a` is an identifier implicitly of type `atm`. This is a binding operation: free occurrences of `a` in the expression *exp* are bound in `new a in` *exp* `end`. Its behaviour is analogous to the Standard ML declaration

$$\texttt{let val a = ref() in } \mathit{exp} \texttt{ end} \tag{6}$$

in that the expression in (5) is evaluated by associating `a` with the first semantic atom unused by the current value environment and then evaluating *exp* in that augmented value environment. (We formulate this more precisely at the end of this section.) As in ML, evaluation in FreshML is done after type checking, and it is there that an important difference between the expressions in (5) and (6) shows up. Compared with ML's type-checking of the expression in (6), the FreshML type system imposes a restriction on the expression in (5) which always seems to be present in uses of 'fresh names' in informal syntax-manipulating algorithms, namely that

> *expressions in the scope of a fresh name* `a` *only use it in ways that are insensitive to renaming.*

For example, although `let val a = ref() in a end` has type `unit ref` in ML, the expression `new a in a end` is not typeable in FreshML—the meaning of `a` is clearly sensitive to renaming `a`. On the other hand, in the next section we introduce atom-abstraction expressions such as `a.a`, whose meaning (either operationally or denotationally) is insensitive to renaming `a` even though they contain free occurrences of `a`, giving rise to well typed expressions such as `new a in a.a end`. (Some other examples of well typed `new`-expressions are given in Fig. 1.)

Type System. To achieve the restrictions mentioned above, the FreshML type system deduces for an expression *exp* judgments not only about its type, $\Gamma \vdash \mathit{exp} : \mathit{ty}$, but also about which atoms `a` are *fresh* with respect to it,

$$\Gamma \vdash \mathit{exp} \# \texttt{a}. \tag{7}$$

[3] In the current experimental version of FreshML, there is only one such type. Future versions will allow the programmer to declare as many distinct copies of this type as needed, for example, for the distinct sorts of names there are in a particular object language.

Here $\mathtt{a}$ is some value identifier assigned type $\mathtt{atm}$ by the typing context Γ. FreshML typing contexts may contain both typing assumptions about value identifiers, $\mathtt{x} : \mathit{ty}$, and freshness assumptions about them, $\mathtt{x} \# \mathtt{a}$ (if Γ contains such an assumption it must also contain $\mathtt{a} : \mathtt{atm}$ and $\mathtt{x} : \mathit{ty}$ for some type ty). The intended meaning of statements such as '$\mathtt{x} \# \mathtt{a}$' is that, in the given value environment, the denotation of $\mathtt{x}$ (an element of an *FM-set*) does not contain the semantic atom associated to $\mathtt{a}$ in its *support*—the mathematical notions of 'FM-set' and 'support' are explained in the Appendix (see Definitions A.1 and A.2). Here we just give rules for inductively generating typing and freshness judgements that are sound for this notion. In fact it is convenient to give an expression's typing and freshness properties simultaneously, using assertions of the form $\mathit{exp} : \mathit{ty} \# \{\mathtt{a}_1, \dots, \mathtt{a}_n\}$ standing for the conjunction

$$\mathit{exp} : \mathit{ty} \ \ \& \ \ \mathit{exp} \# \mathtt{a}_1 \ \ \& \ \ \cdots \ \ \& \ \ \mathit{exp} \# \mathtt{a}_n \ .$$

Thus the FreshML type system can be specified using judgments of the form

$$\Gamma \vdash \mathit{exp} : \mathit{ty} \# \bar{\mathtt{a}} \tag{8}$$

where

$$\Gamma = (\mathtt{x}_1 : \mathit{ty}_1 \# \bar{\mathtt{a}}_1), \dots, (\mathtt{x}_n : \mathit{ty}_n \# \bar{\mathtt{a}}_n) \tag{9}$$

and

- $\mathtt{x}_1, \dots, \mathtt{x}_n$ are distinct value identifiers which include all the free identifiers of the FreshML expression *exp*;
- $\mathit{ty}_1, \dots, \mathit{ty}_n$ and ty are FreshML types[4];
- $\bar{\mathtt{a}}_1, \dots, \bar{\mathtt{a}}_n$ and $\bar{\mathtt{a}}$ are finite sets of value identifiers with the property that each $\mathtt{a} \in \bar{\mathtt{a}}_1 \cup \ldots \cup \bar{\mathtt{a}}_n \cup \bar{\mathtt{a}}$ is assigned type $\mathtt{atm}$ by the typing context, i.e. is equal to one of the $\mathtt{x}_i$ with $\mathit{ty}_i = \mathtt{atm}$.

We write $\Gamma \vdash \mathit{exp} : \mathit{ty}$ as an abbreviation for $\Gamma \vdash \mathit{exp} : \mathit{ty} \# \emptyset$. When discussing just the freshness properties of an expression, we write $\Gamma \vdash \mathit{exp} \# \mathtt{a}$ to mean that $\Gamma \vdash \mathit{exp} : \mathit{ty} \# \{\mathtt{a}\}$ holds for some type ty.

The rule for generating type and freshness information for new-expressions is (11) in Fig. 2. The notation $\mathtt{a} : \mathtt{atm} \otimes \Gamma$ used there indicates the context obtained from Γ by adding the assumptions $\mathtt{a} : \mathtt{atm}$ and $\mathtt{x} \# \mathtt{a}$ for each value identifier $\mathtt{x}$ declared in Γ (where we assume $\mathtt{a}$ does not occur in Γ). For example, if $\Gamma = (\mathtt{x} : \mathtt{atm}), (\mathtt{y} : \mathit{ty} \# \{\mathtt{x}\})$, then

$$\mathtt{a} : \mathtt{atm} \otimes \Gamma = (\mathtt{a} : \mathtt{atm}), (\mathtt{x} : \mathtt{atm} \# \{\mathtt{a}\}), (\mathtt{y} : \mathit{ty} \# \{\mathtt{a}, \mathtt{x}\}) \ .$$

The side condition $\mathtt{a} \notin \mathrm{dom}(\Gamma) \cup \bar{\mathtt{a}}$ in rule (11) is comparable to ones for more familiar binding constructs, such as function abstraction; given Γ and $\bar{\mathtt{a}}$, within the α-equivalence class of the expression `new a in` *exp* `end` we have room to choose the bound identifier $\mathtt{a}$ so that the side condition is satisfied.

[4] In the current experimental version of FreshML, we just consider monomorphic types built up from basic ones like `atm` and `string` using products, functions, recursively defined data type constructions and the atom-abstraction type constructor described in the next section. Introducing type schemes and ML-style polymorphism seems unproblematic in principle, but remains a topic for future work.

$$\frac{(\mathtt{x} : ty \mathrel{\#} \bar{\mathtt{a}}') \in \Gamma \quad \bar{\mathtt{a}} \subseteq \bar{\mathtt{a}}'}{\Gamma \vdash \mathtt{x} : ty \mathrel{\#} \bar{\mathtt{a}}} \tag{10}$$

$$\frac{\mathtt{a} : \mathtt{atm} \otimes \Gamma \vdash exp : ty \mathrel{\#} (\{\mathtt{a}\} \cup \bar{\mathtt{a}}) \quad \mathtt{a} \notin \mathrm{dom}(\Gamma) \cup \bar{\mathtt{a}}}{\Gamma \vdash (\mathtt{new\ a\ in}\ exp\ \mathtt{end}) : ty \mathrel{\#} \bar{\mathtt{a}}} \tag{11}$$

$$\frac{\Gamma \vdash exp : ty \mathrel{\#} (\bar{\mathtt{a}} \smallsetminus \{\mathtt{a}\}) \quad \Gamma(\mathtt{a}) = \mathtt{atm}}{\Gamma \vdash \mathtt{a.}\,exp : \mathtt{[atm]}\,ty \mathrel{\#} \bar{\mathtt{a}}} \tag{12}$$

$$\frac{\Gamma \vdash exp : \mathtt{[atm]}\,ty \mathrel{\#} (\{\mathtt{a}\} \cup \bar{\mathtt{a}}) \quad \Gamma \vdash \mathtt{a} : \mathtt{atm} \mathrel{\#} \bar{\mathtt{a}}}{\Gamma \vdash exp \mathbin{\mathtt{@}} \mathtt{a} : ty \mathrel{\#} \bar{\mathtt{a}}} \tag{13}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp : \mathtt{[atm]}\,ty \mathrel{\#} \bar{\mathtt{a}} \qquad \mathtt{a}, \mathtt{x} \notin \mathrm{dom}(\Gamma) \cup \bar{\mathtt{a}}' \\ (\mathtt{a} : \mathtt{atm} \otimes \Gamma), (\mathtt{x} : ty \mathrel{\#} \bar{\mathtt{a}}) \vdash exp' : ty' \mathrel{\#} (\{\mathtt{a}\} \cup \bar{\mathtt{a}}')\end{array}}{\Gamma \vdash \mathtt{case}\ exp\ \mathtt{of}\ \{\mathtt{a.x}\ \mathtt{=>}\ exp'\} : ty' \mathrel{\#} \bar{\mathtt{a}}'} \tag{14}$$

$$\frac{\Gamma(\mathtt{a}) = \Gamma(\mathtt{b}) = \mathtt{atm} \quad \Gamma \vdash exp : ty \mathrel{\#} \bar{\mathtt{a}} \quad \Gamma[\mathtt{a} \mathrel{\#} \mathtt{b}] \vdash exp' : ty \mathrel{\#} \bar{\mathtt{a}}}{\Gamma \vdash \mathtt{ifeq(a,b)\ then}\ exp\ \mathtt{else}\ exp' : ty \mathrel{\#} \bar{\mathtt{a}}} \tag{15}$$

$$\frac{\Gamma(\mathtt{a}) = \mathtt{atm}\ \text{ for all } \mathtt{a} \in \bar{\mathtt{a}}}{\Gamma \vdash \mathtt{()} : \mathtt{unit} \mathrel{\#} \bar{\mathtt{a}}} \tag{16}$$

$$\frac{\Gamma \vdash exp_1 : ty_1 \mathrel{\#} \bar{\mathtt{a}} \quad \Gamma \vdash exp_2 : ty_2 \mathrel{\#} \bar{\mathtt{a}}}{\Gamma \vdash \mathtt{(}exp_1\mathtt{,}\,exp_2\mathtt{)} : ty_1 \mathbin{\mathtt{*}} ty_2 \mathrel{\#} \bar{\mathtt{a}}} \tag{17}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp : ty_1 \mathbin{\mathtt{*}} ty_2 \mathrel{\#} \bar{\mathtt{a}} \qquad \mathtt{x}, \mathtt{y} \notin \mathrm{dom}(\Gamma) \\ \Gamma, (\mathtt{x} : ty_1 \mathrel{\#} \bar{\mathtt{a}}), (\mathtt{y} : ty_2 \mathrel{\#} \bar{\mathtt{a}}) \vdash exp' : ty' \mathrel{\#} \bar{\mathtt{a}}'\end{array}}{\Gamma \vdash \mathtt{case}\ exp\ \mathtt{of}\ \{\mathtt{(x,y)}\ \mathtt{=>}\ exp'\} : ty' \mathrel{\#} \bar{\mathtt{a}}'} \tag{18}$$

$$\frac{\Gamma \vdash exp : ty \mathrel{\#} \bar{\mathtt{a}} \quad \Gamma, (\mathtt{x} : ty \mathrel{\#} \bar{\mathtt{a}}) \vdash exp' : ty' \mathrel{\#} \bar{\mathtt{a}}' \quad \mathtt{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{let\ val\ x\ =}\ exp\ \mathtt{in}\ exp'\mathtt{end} : ty' \mathrel{\#} \bar{\mathtt{a}}'} \tag{19}$$

$$\frac{\begin{array}{c}\Gamma, (\mathtt{f} : ty \mathbin{\mathtt{->}} ty'), (\mathtt{x} : ty) \vdash exp : ty' \quad \mathtt{f}, \mathtt{x} \notin \mathrm{dom}(\Gamma) \\ \Gamma(\mathtt{a}) = \mathtt{atm}, \text{ for all } \mathtt{a} \in \bar{\mathtt{a}} \qquad \Gamma \vdash \mathtt{x}_i : ty_i \mathrel{\#} \bar{\mathtt{a}}, \text{ for all } \mathtt{x}_i \in \mathrm{fv}(exp) \smallsetminus \{\mathtt{f}, \mathtt{x}\}\end{array}}{\Gamma \vdash \mathtt{fun\ f\ =}\ \{\mathtt{x}\ \mathtt{=>}\ exp\} : (ty \mathbin{\mathtt{->}} ty') \mathrel{\#} \bar{\mathtt{a}}} \tag{20}$$

$$\frac{\Gamma \vdash exp_1 : (ty \mathbin{\mathtt{->}} ty') \mathrel{\#} \bar{\mathtt{a}} \quad \Gamma \vdash exp_2 : ty \mathrel{\#} \bar{\mathtt{a}}}{\Gamma \vdash exp_1\ exp_2 : ty' \mathrel{\#} \bar{\mathtt{a}}} \tag{21}$$

$$\frac{ty \text{ pure} \quad \Gamma \vdash exp : ty \quad \Gamma(\mathtt{a}) = \mathtt{atm}, \text{ for all } \mathtt{a} \in \bar{\mathtt{a}}}{\Gamma \vdash exp : ty \mathrel{\#} \bar{\mathtt{a}}} \tag{22}$$

Fig. 2. Excerpt from the type system of FreshML

To understand rule (11) better, consider the special case when Γ and $\bar{\mathtt{a}}$ are empty. This tells us that a closed expression of the form `new a in` *exp* `end` has type *ty* if not only $\mathtt{a} : \mathtt{atm} \vdash \mathit{exp} : \mathit{ty}$, but also $\mathtt{a} : \mathtt{atm} \vdash \mathit{exp} \mathrel{\#} \mathtt{a}$. This second property guarantees that although *exp* may involve the atom `a`, its meaning is unchanged if we rename `a`—and hence it is meaningful to 'anonymise' `a` in *exp* by forming the expression `new a in` *exp* `end`.

But how do we generate the freshness assertions needed in the hypothesis of rule (11) in the first place? One rather trivial source arises from the fact that if `a` is fresh for all the free identifiers in an expression *exp*, then it is fresh for *exp* itself (this is a derivable property of the FreshML type system); so in particular `a` is always fresh for closed expressions. However, it can indeed be the case that $\Gamma \vdash \mathit{exp} \mathrel{\#} \mathtt{a}$ holds with `a` occurring freely in *exp*. Section 3 introduces the principal source of such non-trivial instances of freshness.

Operational Semantics. At the beginning of this section we said that the operational behaviour of the FreshML expression `new a in` *exp* `end` is like that of the ML expression `let val a = ref() in` *exp* `end`. In order to be more precise, we need to describe the operational semantics of FreshML. The current experimental version of FreshML is a pure functional programming language, in the sense that the only effects of expression evaluation are either to produce a value or to not terminate. We describe evaluation of expressions *exp* using a judgement of the form

$$E \vdash \mathit{exp} \Rightarrow v$$

where E is a *value environment* (whose domain contains the free value identifiers of *exp*) and v is a *semantic value*. These are mutually recursively defined as follows: E is a finite mapping from value identifiers to semantic values; and, for the fragment of FreshML typed in Fig. 2, we can take v to be given by the grammar

$$v ::= a \mid \mathit{abs}(a, v) \mid \mathit{unit} \mid \mathit{pr}(v, v) \mid \mathit{fun}(\mathtt{x}, \mathtt{x}, \mathit{exp}, E)$$

where a ranges over semantic atoms, `x` over value identifiers, and *exp* over expressions. The rule for evaluating `new`-expressions is (24) in Fig. 3. The notation $\mathit{fa}(E)$ used in that rule stands for the finite set of 'synthetically' free semantic atoms of a value environment E; this is defined by

$$\mathit{fa}(E) \;=\; \bigcup\nolimits_{\mathtt{x} \in \mathrm{dom}(E)} \mathit{fa}(E(\mathtt{x}))$$

where $\mathit{fa}(a) = \{a\}$, $\mathit{fa}(\mathit{abs}(a, v)) = \mathit{fa}(v) \smallsetminus \{a\}$, $\mathit{fa}(\mathit{pr}(v_1, v_2)) = \mathit{fa}(v_1) \cup \mathit{fa}(v_2)$, $\mathit{fa}(\mathit{unit}) = \emptyset$ and $\mathit{fa}(\mathit{fun}(\mathtt{f}, \mathtt{x}, \mathit{exp}, E)) = \bigcup\nolimits_{\mathtt{y} \in \mathrm{fv}(\mathit{exp}) \smallsetminus \{\mathtt{f}, \mathtt{x}\}} \mathit{fa}(E(\mathtt{y}))$.

It should be emphasised that the rules in Fig. 3 are only applied to well-typed expressions. Evaluation preserves typing and freshness information in the following sense.

Theorem 2.1 (Soundness of the type system with respect to evaluation). *If $\Gamma \vdash \mathit{exp} : \mathit{ty} \mathrel{\#} \bar{\mathtt{a}}$, $E \vdash \mathit{exp} \Rightarrow v$ and $E : \Gamma$, then $v : \mathit{ty}$ and $\mathit{fa}(v) \cap \{\, E(\mathtt{a}) \mid \mathtt{a} \in \bar{\mathtt{a}} \,\} = \emptyset$. (The definitions of '$E : \Gamma$' and '$v : \mathit{ty}$' for the fragment of FreshML typed in Fig. 2 are given in Fig. 4)*

$$\frac{E(\mathtt{x}) = v}{E \vdash \mathtt{x} \Rightarrow v} \tag{23}$$

$$\frac{E[\mathtt{a} \mapsto a] \vdash \mathit{exp} \Rightarrow v \quad a \notin \mathit{fa}(E)}{E \vdash \mathtt{new\ a\ in}\ \mathit{exp}\ \mathtt{end} \Rightarrow v} \tag{24}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \quad E(\mathtt{a}) = a}{E \vdash \mathtt{a.}\mathit{exp} \Rightarrow \mathit{abs}(a, v)} \tag{25}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow \mathit{abs}(a', v') \quad E(\mathtt{a}) = a \quad v = (a'\ a) \cdot v'}{E \vdash \mathit{exp}\ \mathtt{@}\ \mathtt{a} \Rightarrow v} \tag{26}$$

$$\frac{\begin{array}{c} E \vdash \mathit{exp} \Rightarrow \mathit{abs}(a, v) \qquad a' \notin \mathit{fa}(E) \cup \mathit{fa}(\mathit{abs}(a, v)) \\ v'' = (a'\ a) \cdot v \qquad E[\mathtt{a} \mapsto a', \mathtt{x} \mapsto v''] \vdash \mathit{exp}' \Rightarrow v' \end{array}}{E \vdash \mathtt{case}\ \mathit{exp}\ \mathtt{of}\ \{\mathtt{a.x \Rightarrow}\ \mathit{exp}'\} \Rightarrow v'} \tag{27}$$

$$\frac{E(\mathtt{a}) = E(\mathtt{b}) \quad E \vdash \mathit{exp} \Rightarrow v}{E \vdash \mathtt{ifeq(a,b)\ then}\ \mathit{exp}\ \mathtt{else}\ \mathit{exp}' \Rightarrow v} \tag{28}$$

$$\frac{E(\mathtt{a}) \neq E(\mathtt{b}) \quad E \vdash \mathit{exp}' \Rightarrow v'}{E \vdash \mathtt{ifeq(a,b)\ then}\ \mathit{exp}\ \mathtt{else}\ \mathit{exp}' \Rightarrow v'} \tag{29}$$

$$E \vdash \mathtt{()} \Rightarrow \mathit{unit} \tag{30}$$

$$\frac{E \vdash \mathit{exp}_1 \Rightarrow v_1 \quad E \vdash \mathit{exp}_2 \Rightarrow v_2}{E \vdash \mathtt{(}\mathit{exp}_1\mathtt{,}\mathit{exp}_2\mathtt{)} \Rightarrow \mathit{pr}(v_1, v_2)} \tag{31}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow \mathit{pr}(v_1, v_2) \quad E[\mathtt{x} \mapsto v_1, \mathtt{y} \mapsto v_2] \vdash \mathit{exp}' \Rightarrow v'}{E \vdash \mathtt{case}\ \mathit{exp}\ \mathtt{of}\ \{\mathtt{(x,y) \Rightarrow}\ \mathit{exp}'\} \Rightarrow v'} \tag{32}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \quad E[\mathtt{x} \mapsto v] \vdash \mathit{exp}' \Rightarrow v'}{E \vdash \mathtt{let\ x\ =}\ \mathit{exp}\ \mathtt{in}\ \mathit{exp}'\ \mathtt{end} \Rightarrow v'} \tag{33}$$

$$E \vdash \mathtt{fun\ f\ =}\ \{\mathtt{x \Rightarrow}\ \mathit{exp}\} \Rightarrow \mathit{fun}(\mathtt{f}, \mathtt{x}, \mathit{exp}, E) \tag{34}$$

$$\frac{\begin{array}{c} E \vdash \mathit{exp}_1 \Rightarrow v_1 \qquad E \vdash \mathit{exp}_2 \Rightarrow v_2 \\ v_1 = \mathit{fun}(\mathtt{f}, \mathtt{x}, \mathit{exp}, E_1) \qquad E_1[\mathtt{f} \mapsto v_1, \mathtt{x} \mapsto v_2] \vdash \mathit{exp} \Rightarrow v \end{array}}{E \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow v} \tag{35}$$

Fig. 3. Excerpt from the operational semantics of FreshML

$$\frac{a \in \mathbb{A}}{a : \mathtt{atm}} \qquad \frac{a \in \mathbb{A} \quad v : ty}{abs(a, v) : \mathtt{[atm]}\,ty} \qquad unit : \mathtt{unit} \qquad \frac{v_1 : ty_1 \quad v_2 : ty_2}{pr(v_1, v_2) : ty_1 \mathbin{\mathtt{*}} ty_2}$$

$$\frac{\Gamma, (\mathtt{f} : ty \mathbin{\mathtt{->}} ty'), (\mathtt{x} : ty) \vdash exp : ty \mathbin{\mathtt{->}} ty' \quad \mathtt{f}, \mathtt{x} \notin \mathrm{dom}(\Gamma) \quad E : \Gamma}{fun(\mathtt{f}, \mathtt{x}, exp, E) : ty \mathbin{\mathtt{->}} ty'}$$

$$\frac{\begin{array}{c}\mathrm{dom}(E) = \mathrm{dom}(\Gamma) \\ E(\mathtt{x}_i) : ty_i \text{ and } fa(E(\mathtt{x}_i)) \cap \{\, E(\mathtt{a}) \mid \mathtt{a} \in \bar{\mathtt{a}}_i \,\} = \emptyset, \text{ for all } (\mathtt{x}_i : ty_i \mathbin{\#} \bar{\mathtt{a}}_i) \in \Gamma\end{array}}{E : \Gamma}$$

Fig. 4. Typing semantics values and value environments

3 Atom Abstraction

FreshML user-declared data types can have value constructors involving binding, via a type constructor `[atm](−)` for *atom abstractions*. The data type `lam` declared in Fig. 1 provides an example of this, with its constructor `Lam : [atm]lam -> lam`. Expressions of an atom abstraction type `[atm]`*ty* are introduced with a syntactic form which is written `a.`*exp*, where `a` is a value identifier of type `atm` and *exp* an expression of type *ty*. Such *atom abstraction expressions* behave like pairs in which the first component is hidden, in a way comparable to hiding in abstract data types [13]. The operations for accessing the second component are discussed in Sects 4 and 5. We claim that two such expressions, `a.`*exp* and `a'.`*exp'*, are contextually equivalent (i.e. are interchangeable in any complete FreshML program without affecting the observable results of evaluating it) if and only if

> *for some (any) fresh* $\mathtt{a}''$*,* $(\mathtt{a}''\,\mathtt{a}) \cdot exp$ *and* $(\mathtt{a}''\,\mathtt{a}') \cdot exp'$ *are contextually equivalent expressions of type ty*

where $(\mathtt{a}''\,\mathtt{a}) \cdot exp$ indicates the expression obtained by interchanging all occurrences of $\mathtt{a}''$ and `a` in *exp*. It is for this reason that values of type `lam` correspond to α-equivalence classes of lambda terms: see [7, Theorem 2.1].

Atom abstraction expressions `a.`*exp* are evaluated using rule (25) in Fig. 3; and their typing and freshness properties are given by rule (12) in Fig. 2. In that rule, the notation $\bar{\mathtt{a}} \smallsetminus \{\mathtt{a}\}$ means the finite set $\{\, \mathtt{a}' \in \bar{\mathtt{a}} \mid \mathtt{a}' \neq \mathtt{a} \,\}$; and the side-condition $\Gamma(\mathtt{a}) = \mathtt{atm}$ means that, with Γ as in equation (9), $\mathtt{a} = \mathtt{x}_i$ for some i with $ty_i = \mathtt{atm}$. To understand rule (12) better, consider the special case when $\bar{\mathtt{a}} = \{\mathtt{a}\}$: then the rule tells us that provided $\Gamma(\mathtt{a}) = \mathtt{atm}$ and *exp* is typeable in context Γ, then `a` *is always fresh for* `a.`*exp*, i.e. $\Gamma \vdash (\mathtt{a.}exp) \mathbin{\#} \mathtt{a}$. This is the principal source of freshness assertions in FreshML. For example:

Example 3.1. Given the declarations in Fig. 1 and some straightforward rules for typing data type constructors (which we omit, but which are analogous to the rules (16) and (17) for unit and pairs in Fig. 2), from rule (12) we have

$$\mathtt{a} : \mathtt{atm} \vdash (\mathtt{a.(Var\ a)}) : \mathtt{[atm]lam} \mathbin{\#} \{\mathtt{a}\}$$

and then

$$\mathtt{a} : \mathtt{atm} \vdash (\mathtt{Lam\,a.(Var\ \ a)}) : \mathtt{lam} \# \{\mathtt{a}\}.$$

Applying rule (11) to this yields

$$\vdash (\mathtt{new\ a\ in\ Lam\,a.(Var\ \ a)\ end}) : \mathtt{lam}.$$

This closed expression of type `lam` is a FreshML representation of the lambda term $\lambda a.a$.

Note that *atom abstraction is not a binder* in FreshML: the free identifiers of `a.`*exp* are `a` and all those of *exp*. The syntactic restriction that the expression to the left of the 'abstraction dot' be an identifier is needed because we only consider freshness assertions '*exp* # `a`' with `a` an identifier rather than a compound expression (in order to keep the type system as simple as possible). This does not really restrict the expressiveness of FreshML, since a more general form of atom abstraction '*atexp*`.`*exp*' (with *atexp* of type `atm`) can be simulated with the `let`-expression `let val a =` *atexp* `in a.`*exp* `end`. (The typing rule for `let`-expressions is (19) in Fig. 2.)

Remark 3.2 (binding = renameability + name hiding). Example 3.1 illustrates the fact that, unlike metalanguages that represent object-level binding via lambda abstraction, FreshML separates the renaming and the hiding aspects of variable binding. On the one hand `a` is still a free identifier in `a.`*exp*, but on the other hand the fact that `new a in` $-$ `end` is a statically scoped binder can be used to hide the name of an atom (subject to the freshness conditions discussed in Sect. 2). We illustrate why this separation of the renaming and the hiding aspects of variable binding can be convenient in Example 4.1 below. To give the example we first have to discuss mechanisms for computing with expressions of atom abstraction type `[atm]`*ty*. FreshML offers two related alternatives: *concretion* expressions, *exp* `@ a`, and `case`-expressions using *abstraction patterns*, such as `case` *exp* `of {a.x =>` *exp'*`}`. We discuss each in turn.

4 Concretion

Values of atom abstraction type have a double nature. So far we have seen their pair-like aspect; but as noted in [7, Lemma 4.1], they also have a function-like aspect: we can choose the name `a` of the first component in an atom abstraction *exp* : `[atm]`*ty* as we like, subject to a certain freshness restriction, and then the second component turns out to be a function of that choice, which we write as `a` $\mapsto$ *exp* `@ a`. We call *exp* `@ a` the *concretion*[5] of the atom abstraction *exp* at the atom `a`. The typing and freshness properties of concretions are given by rule (13) in Fig. 2. Note in particular (taking $\overline{\mathtt{a}} = \emptyset$ in the rule) that given $\Gamma \vdash$ *exp* : `[atm]`*ty*, in order to deduce $\Gamma \vdash$ *exp* `@ a` : *ty* we need to know not only that $\Gamma(\mathtt{a}) = \mathtt{atm}$, but also that $\Gamma \vdash$ *exp* # `a`. The denotational justification for

[5] The terminology is adopted from [11, Sect. 12.1].

this is given by Proposition A.2 of the Appendix. Operationally, the behaviour of concretion is given by rule (26) in Fig. 3. Thus evaluation of *exp* @ a proceeds by first evaluating *exp*; if a semantic value of the form $abs(a', v')$ is returned and the semantic atom associated with a in the current value environment is a, then the result of evaluating *exp* @ a is the semantic value $(a'\ a) \cdot v'$ obtained from v' by interchanging all occurrences of a' and a in v'. By analogy with β-conversion for λ-abstraction and application, it is tempting to replace the use of transposition $(a'\ a)\cdot(-)$ by substitution $[a/a'](-)$, but this would not be correct. The reason for this has to do with the fact that while a.(−) is used to represent binding in object languages, it is not itself a binding operation in FreshML; so the substitution $[a/a'](-)$ can give rise to capture at the object-level in a way which $(a'\ a) \cdot (-)$ cannot. Here is an example to illustrate this: the result of evaluating

$$(\mathtt{a}'.(\mathtt{a}.(\mathtt{Var}\ \mathtt{a}')))\ @\ \mathtt{a}$$

in the value environment $E = \{\mathtt{a} \mapsto a, \mathtt{a}' \mapsto a'\}$ is $abs(a', \mathtt{Var}\ a)$. Using $[a/a'](-)$ instead of $(a'\ a) \cdot (-)$ one would obtain the wrong value, namely $abs(a, \mathtt{Var}\ a)$, which is semantically distinct from $abs(a', \mathtt{Var}\ a)$ (in as much as the two semantic values have different denotations in the FM-sets model—see Sect. A.4).

Here is an example combining atom abstraction, concretion and local freshness expressions (together with standard case-expressions and function declaration).

Example 4.1. One of the semantic properties of the atom abstraction set-former in the model in [7] (and the related models in [5]) which distinguish it from function abstraction is that it commutes with disjoint unions up to natural bijection. We can easily code this bijection in FreshML as follows.

```
(* A type constructor for disjoint unions. *)
datatype ('a,'b)sum = Inl of 'a | Inr of 'b;
(* A bijection i:[atm](('a,'b)sum) -> ([atm]'a,[atm]'b)sum. *)
fun i = { e => new a in
                 case e@a of
                   { Inl x => Inl(a.x)
                   | Inr y => Inr(a.y) }
               end }
```

This illustrates the use of the fact mentioned in Remark 3.2 that name abstraction and name hiding are separated in FreshML: note that in the definition of i, the locally fresh atom a is not used in an abstraction immediately, but rather at two places nested within its scope (a.x and a.y).

The bijection in this example can be coded even more perspicuously using pattern matching:

```
fun i' = { a.(Inl x) => Inl(a.x)
         | a.(Inr y) => Inr(a.y) }
```

Expressions like 'a.(Inl x)' are *abstraction patterns*. The fact that there is a useful matching mechanism for them is one of the major innovations of FreshML and we discuss it next.

5 Matching with Atom Abstraction Patterns

It is not possible to split semantic values of type `[atm]`*ty* into (atom,value)-pairs uniquely, because given $abs(a, v)$ then for any $a' \notin fa(v)$, $abs(a, v)$ has the same denotation as $abs(a', (a'\ a) \cdot v)$. However, if we only use the second component $(a'\ a) \cdot v$ in a way that is insensitive to which particular fresh a' is chosen, we get a well-defined means of specifying a function on atom abstractions via matching against an *abstraction pattern*. The simplest example of such a pattern takes the form `a.x`, where `a` and `x` are distinct identifiers. Rule (14) in Fig. 2 gives the typing and freshness properties and rule (27) in Fig. 3 the evaluation properties for a `case`-expression with a single match using such a pattern. (In the expression `case` *exp* `of` {`a.x =>` *exp*′}, the distinct identifiers `a` and `x` are binders with *exp*′ as their scope.)

Figure 1 gives some examples of declarations involving more complicated, nested abstraction patterns. We omit the formal definition of matching against such patterns, but the general idea is that atom identifiers to the left of an 'abstraction dot' in a pattern represent semantic atoms that are fresh in the appropriate sense; and by checking freshness assertions, the type system ensures that the expression to the right of '`=>`' in a match uses such identifiers in a way that is insensitive to renaming. For example, this implicit freshness in matching is what ensures that `sub` in Fig. 1 implements *capture-avoiding* substitution—in the last match clause, `b` is automatically fresh for `t` and so it makes sense to apply the substitution function `sub(t, a.−)` under `Lam b.−`. Another example is the declaration `bv` in Fig. 1, which does not type check because in the last match clause `a` is not fresh for `a::(bv t)`.

In the current experimental version of FreshML, all uses of abstraction patterns are eliminated by macro-expanding them using concretion and local freshness. For example, as rules (14) and (27) may suggest, `case` *exp* `of` {`a.x =>` *exp*′} can be regarded as an abbreviation for

`new a′ in case` *exp* `@ a′ of` {`x =>` [a′/a]*exp*′} `end`

(where $\mathtt{a}' \notin \mathrm{fv}(exp)$). However, to accommodate the more general notions of abstraction mentioned at the end of Sect. 9, we expect that matching with abstraction patterns will have to be a language primitive.

Remark 5.1 (Comparison with Standard ML). According to its Definition [12], in Standard ML during type checking a pattern *pat* elaborates in the presence of a typing context Γ to a piece of typing context Γ' (giving the types of the identifiers in the pattern) and a type *ty* (the overall type of the pattern); then a match *pat* `=>` *exp*′ elaborates in the context Γ to a function type *ty* `->` *ty*′ if *exp*′ has type *ty*′ in the augmented context $\Gamma \oplus \Gamma'$. Pattern elaboration is a little more complicated in FreshML. For abstraction patterns generate not only a piece of typing context Γ', but also two kinds of freshness assumptions: ones that modify Γ (cf. the use of $\mathtt{a} : \mathtt{atm} \otimes \Gamma$ in rule (14)); and ones that impose freshness restrictions on *exp*′ in a match *pat* `=>` *exp*′ (cf. the use of $exp' : ty' \ \#\ (\{\mathtt{a}\} \cup \overline{\mathtt{a}}')$ in rule (14)).

Example 5.2. In Example 4.1 we gave an example where the use of abstraction patterns allows a simplification compared with code written just using the combination of `new`-expressions and concretion. Sometimes the reverse is the case. For example, in informal practice when specifying a function of finitely many abstractions, it is convenient to use the *same* name for the abstracted variable (and there is no loss of generality in doing this, up to α-conversion). This is not possible using FreshML patterns because, as in ML, we insist that they be *linear*: an identifier must occur at most once in a pattern. However, it is possible through explicit use of a locally fresh atom. Here is a specific example.

In the FM-sets model (see the Appendix), the atom abstraction set-former commutes with cartesian products up to natural bijection. We can code this bijection in FreshML using pattern-matching as follows.

```
(* A bijection ([atm]'a)*([atm]'b) -> [atm]('a * 'b)  *)
fun p1 = { (a.x, b.y) => b.((a.x)@b, y) }
```

Better would be

```
fun p2 = { (e, b.y) => b.(e@b, y) }
```

but an arguably clearer declaration (certainly a more symmetric one) uses local freshness explicitly:

```
fun p3 = { (e, f) => new a in a.(e@a, f@a) end }
```

Simplest of all would be the declaration

```
fun p4 = { (a.x, a.y) => a.(x, y) }
```

but this is not legal, because `(a.x, a.y)` is not a linear pattern. As we discuss in the next section, `atm` is an equality type; so matching patterns with repeated occurrences of identifiers of that type is meaningful (although patterns like `(a.x, a.y)` involve a further complication, in that the repeated identifier is in a 'negative position', i.e. to the left of the 'abstraction dot'). We have insisted on linear patterns in the current version of FreshML in order not to further complicate a notion of matching which, as we have seen, is already more complicated than in ML.

6 Atom Equality

Algorithms for manipulating syntax frequently make use of the decidability of equality of names (of object level variables, for example). Accordingly, the type `atm` of atoms in FreshML admits equality. In particular if *atexp* and *atexp′* are two expressions of type `atm`, then `eq(`*atexp*, *atexp′*`)` is a boolean expression which evaluates to *true* if *atexp* and *atexp′* evaluate to the same semantic atom and evaluates to *false* if they evaluate to different ones. What more need one say? In fact, when it comes to type checking there is more to say. To see why, consider the following declaration of a function taking a list of atoms with one atom abstracted and removing it.

```
fun rem2 = { e => new a in
                   case e@a of
                     { nil => nil
                     | b::bs => ifeq(b,a) then (rem2 a.bs)
                                else b::(rem2 a.bs) }
                   end }
```

This makes use of a form of conditional

`ifeq(a,b) then` *exp* `else` *exp′*

which branches on the equality of `a` and `b`—see rules (28) and (29) in Fig. 3. For the above declaration of `rem2` to type-check as a function `[atm](atm list) -> (atm list)`, one has to apply rule (11) to the subphrase `new a in` ... `end`. Amongst other things, this requires one to prove

(`ifeq(b,a) then (rem2 a.bs) else b::(rem2 a.bs)`) # `a`

in a typing context whose only freshness assumptions are `rem2` # `a` and `e` # `a`. For this to be possible we have to give a typing rule for `ifeq(a,b) then` – `else` – which, while checking the second branch of the conditional, adds the semantically correct information `a` # `b` to the typing context. Such a typing rule is (15) in Fig. 2. (This uses the notation Γ[`a` # `b`] to indicate the typing context obtained from Γ by adding the assumption `a` # `b`; we omit the straightforward formal definition of this for the typing contexts defined as in equation (9).) Although we take account of the fact that `a` and `b` denote distinct atoms when checking the second branch of the conditional, we have not found a need in practice to take account of the fact that they denote the same atom when checking the first branch (by strengthening the second hypothesis of this rule to $[\mathtt{a}/\mathtt{b}](\Gamma \vdash exp : ty \# \bar{\mathtt{a}})$, for example.)

To get information about atom equality to where it is needed for type checking, FreshML also permits the use of *guarded patterns*

pat `where a = b` and *pat* `where a # b`

where `a` and `b` are value identifiers of type `atm`. Such guards are inter-definable with the `ifeq( – , – ) then – else –` construct, but often more convenient. Figure 1 gives several examples of their use. We omit the precise definition of matching such guarded patterns. Note also that the atom equality test expression `eq(`*atexp*, *atexp′*`)` can be regarded as a macro for

```
let val a = atexp in
    let val b = atexp′ in
        ifeq(a,b) then true else false
    end
end .
```

7 Functions

Recall from Sect. 2 that the intended meaning of freshness assertions in FreshML has to do with the notion of the 'support' of an element of an FM-set (see Definition A.1 in the Appendix). The nature of this notion is such that in any reasonable (recursively presented) type system for FreshML, the provable freshness assertions will always be a proper subset of those which are satisfied by the FM-sets model. This is because of the logical complexity of the statement

> 'the semantic atom associated with the identifier `a` is not in the support of the denotation of the function expression `fn{x =>` *exp*`}`'

which involves extensional equality of mathematical functions. So if 'provable freshness' only gives an approximation to 'not in the support of', we should expect that not every denotationally sensible expression will receive a type. (Of course, such a situation is not uncommon for static analyses of properties of functional languages.)

What approximation of the support of a function should be used to infer sound freshness information for function expressions in FreshML? We certainly want type-checking to be decidable. In the current experimental version of FreshML, we take a simple approach (which does ensure decidability) making use of the following general property of freshness

> *if* `a` *is fresh for all the free identifiers in an expression exp, then it is fresh for exp itself*

which is certainly sound for the denotational semantics in **FM-Set**. Applying this in the case when *exp* is a recursive function expression

$$\texttt{fun f = \{x =>}\ \mathit{exp}\texttt{\}} \tag{36}$$

we arrive the typing rule (20) given in Fig. 2. As usual, free occurrences of `f` and `x` in *exp* become bound in the expression (36). We can regard non-recursive function expressions `fn{x =>` *exp*`}` as the special case of expression (36) in which $\texttt{f} \notin \mathrm{fv}(\mathit{exp})$.

Example 7.1. Consider the following FreshML declaration of a function `fv2` for computing the list (possibly with repetitions) of free variables of a lambda term encoded as a value of the data type `lam` in Fig. 1.

```
fun fv2 =
 { Var a => a::nil
 | App(t,u) => append(fv2 t)(fv2 u)
 | Lam a.t => remove a (fv2 t) }
```

This uses auxiliary functions `append` for joining two lists, and `remove : atm -> (atm list) -> (atm list)` for removing an atom from a list of atoms (using the fact that `atm` is an equality type), whose standard definitions we omit.

One might hope that `fv2` is assigned type `lam->(atm list)`, but the current FreshML type system rejects it as untypeable. The problem is the last match clause, `Lam a.t => remove a (fv2 t)`. As explained in Sect. 5, for this to type check we have to prove

$$(\mathtt{a} : \mathtt{atm}), (\mathtt{fv2} : (\mathtt{lam} \texttt{->} (\mathtt{atm\ list})) \# \{\mathtt{a}\}), (\mathtt{e} : [\mathtt{atm}]\mathtt{lam} \# \{\mathtt{a}\}) \\ \vdash \mathtt{remove\,a\,(fv2(e@a))} : \mathtt{atm\ list} \# \{\mathtt{a}\} \qquad (37)$$

This is denotationally correct, because the denotation of `remove` maps a semantic atom a and a list of semantic atoms as to the list $as \setminus \{a\}$; and the support of the latter consists of all the semantic atoms in the list as that are not equal to a. However, the typing rules in Fig. 2 are not sufficiently strong to deduce (37).

It seems that this problem, and others like it, can be solved by using a richer notion of type in which expressions like '$ty \# \overline{\mathtt{a}}$' become first-class types which can be mixed with the other type constructors (in particular, with function types). We have made some initial investigations into the properties of such richer type systems (and associated notions of subtyping induced by making $ty \# \overline{\mathtt{a}}$ a subtype of ty), but much remains to be done. However, for this particular example there is a simple work-around which involves making better use of atom-abstractions and the basic fact (12) about their support. Thus the declaration of `fv` in Fig. 1 makes use of the auxiliary function `rem : [atm](atm list) -> (atm list)` for which

$$(\mathtt{a} : \mathtt{atm}), (\mathtt{fv} : (\mathtt{lam} \texttt{->} (\mathtt{atm\ list})) \# \{\mathtt{a}\}), (\mathtt{e} : [\mathtt{atm}]\mathtt{lam} \# \{\mathtt{a}\}) \\ \vdash (\mathtt{rem\,a.(fv(e@a))}) : \mathtt{atm\ list} \# \{\mathtt{a}\}$$

can be deduced using (12). It follows that `fv` does yield a function of type `lam -> (atm list)` (and its denotation is indeed the function returning the list of free variables of a lambda term).

Remark 7.2. Note that freshness is not a 'logical relation': just because a function maps all arguments not having a given atom in their support to results not having that atom in their support, it does not follow that the atom is fresh for the function itself. Thus the following rule is unsound.

$$\frac{\Gamma, (\mathtt{f} : (ty \texttt{->} ty') \# \overline{\mathtt{a}}), (\mathtt{x} : ty \# \overline{\mathtt{a}}) \vdash exp : ty' \# \overline{\mathtt{a}} \qquad \mathtt{f}, \mathtt{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{fun\ f = \{x => } exp\mathtt{\}} : (ty \texttt{->} ty') \# \overline{\mathtt{a}}} \ \text{(wrong!)}$$

To see this, consider the following example. From rule (12) we have

$$\mathtt{a} : \mathtt{atm}, \mathtt{x} : \mathtt{atm} \# \{\mathtt{a}\} \vdash \mathtt{a.x} : [\mathtt{atm}]\mathtt{atm} \# \{\mathtt{a}\}$$

and so using the above rule (wrong!) we would be able to deduce

$$\mathtt{a} : \mathtt{atm} \vdash \mathtt{fn\{x => a.x\}} : (\mathtt{atm} \texttt{->} [\mathtt{atm}]\mathtt{atm}) \# \{\mathtt{a}\}.$$

This is denotationally incorrect, because the denotation of `fn{x => a.x}` does contain the denotation of `a` in its support. Correspondingly, the operational behaviour of this function expression depends on which semantic atom is associated with `a`: if we replace `a` by `a'` in `fn{x => a.x}`, then under the assumption $\texttt{a} \# \texttt{a}'$ we get a contextually inequivalent function expression, `fn{x => a'.x}`. For applying these function expressions to the same argument `a` yields contextually inequivalent results (in the first case an expression equivalent to `a.a` and in the second case one equivalent to `a'.a`).

8 Purity

Consider the following declaration of a function `count` for computing the number of lambda abstractions in a lambda term encoded as a value of the data type `lam` in Fig. 1.

```
fun count =
 { Var a => 0
 | App(t,u) => (count t)+(count u)
 | Lam a.t => (count a)+1 }
```

For this to type check as a function `lam -> int`, the last match clause requires

$$(\texttt{a} : \texttt{atm}), (\texttt{count} : (\texttt{lam -> int}) \# \{\texttt{a}\}), (\texttt{t} : \texttt{lam}) \\ \vdash (\texttt{(count t)+1}) : \texttt{int} \# \{\texttt{a}\}$$

to be proved. The interpretation of this judgement holds in the FM-sets model because the denotation of `int` is a 'pure' FM-set, i.e. one whose elements all have empty support. Accordingly, we add rule (22) in Fig. 2 to the FreshML type system; using it, we do indeed get that `count` has type `lam -> int`. The condition '*ty* pure' in the hypothesis of this rule is defined by induction on the structure of the type *ty* and amounts to saying that *ty* does not involve `atm` or `->`[6] in its construction.

The current experimental version of FreshML is a pure functional programming language, in the sense that the only effects of expression evaluation are either to produce a value or to not terminate. This has an influence on the soundness of rule (22). For example, if we add to the language an exception mechanism in which exception packets contain values involving atoms, then it may no longer be the case that an integer expression *exp* : `int` satisfies *exp* # `a` for *any* atom `a`. To restore the soundness of rule (22) in the presence of such computational effects with non-trivial support, one might consider imposing a 'value restriction', by insisting that the rule only applies to expressions *exp* that are non-expansive in the sense of [12, Sect. 4.7]. However, note that the rule (19) for `let`-expressions rather undoes such a value-restriction. For using rule (19), the freshness properties of *exp* which we could have deduced from the unrestricted rule (22) can be deduced for the semantically equivalent expression `let val x =` *exp* `in x end` from the value-restricted version. This highlights

[6] For Theorem 2.1 to hold, in rule (22) we need that *ty* pure implies $fa(v) = \emptyset$ for any semantic value v of type *ty*; excluding the use of function types (as well as `atm`) in pure types is a simple way of ensuring this.

the fact that the soundness of rule (19), and also rules (14) and (18) in Fig. 2, depends upon the evaluation of *exp* not producing an effect with non-empty support. Should one try to restrict these rules as well? Probably it is better to curtail the computational effects. For example, although it is certainly desirable to add an exception-handling mechanism to the current version of the language, it may be sufficient to have one which only raises packets containing values with empty support (character strings, integers, etc). Investigation of this is a matter for future work—which brings us to our final section.

9 Related and Future Work

Related work

The model presented in [7] was one of three works on the metamathematics of syntax with binders using categories of (pre)sheaves which appeared simultaneously in 1999—the other two being [5] and [9]. The starting point for these works is a discovery, made independently by several of the authors, which can be stated roughly as follows.

> *The quotient by α-equivalence of an inductively defined set of abstract syntax trees (for some signature involving binders) can be given an initial algebra semantics provided one works with initial algebras of functors not on sets, but on categories of 'variable' sets, i.e. certain categories of sheaves or presheaves.*

There is a strong connection between initial algebras for functors and recursive data types, so this observation should have some consequences for programming language design, and more specifically, for new forms of user declared data type. That is what we have investigated here. For us, the notion of (finite) *support*, which is a key feature of the model in [7], was crucial—giving rise as it does to FreshML's idiom of computing with freshness. While the presheaf models considered in [5] have weaker notions of support (or 'stage') than does the FM-sets model, it seems that they too can model languages with notions of abstraction similar to the one in FreshML (Plotkin, private communication).

Miller [10] proposed tackling the problems motivating the work in this paper by incorporating the techniques of *higher order abstract syntax*, HOAS [16], into an ML-like programming language, ML_λ, with intentional function types *ty* `=>` *ty′*. Compared with HOAS, the approach in [7] and [5] is less ambitious in what it seeks to lift to the metalevel: like HOAS we promote object-level renaming to the metalevel, but unlike HOAS we leave object-level substitution to be defined case-by-case using structural recursion. The advantage is that FreshML data types using `[atm]`*ty* retain the pleasant recursion/induction properties of classical first order algebraic data types: see Sect. 5 of [7]. It is also the case that names of bound variables are inaccessible to the ML_λ programmer, whereas they are accessible to a FreshML programmer in a controlled way.

Formal properties of FreshML

In this paper we have mostly concentrated on explaining the ideas behind our approach and giving examples, rather than on presenting the formal properties of the language. Such properties include:

1. Decidability of type/freshness inference.
2. Correspondence results connecting the operational and denotational semantics.
3. The correctness of the encoding of the set of terms over a 'binding signature' [5, Sect. 2], modulo renaming of bound variables, as a suitable FreshML data type. For example, values of type `lam` in Fig. 1 modulo contextual equivalence (or equality of denotations) correspond to α-equivalence classes of untyped lambda terms, with free identifiers of type `atm` corresponding to free variables.
4. Transfer of principles of structural induction for inductively defined FM-sets involving atom abstraction (cf. [7, Sect. 5]) to induction principles for FreshML data types. More generally, the 'И-quantifier' introduced in [7] should feature in an LCF-style program logic for FreshML.

Some of the details will appear in the second author's forthcoming PhD thesis [6]. At this stage, just as important as proving such properties is accumulating experience with programming in FreshML, to see if the idiom it provides for metaprogramming with bound names modulo renaming is a useful one.

Future work

To improve the usefulness of FreshML for programming with bound names, it is already clear that we must investigate richer syntactic forms of abstraction. At the moment we permit abstraction over a single type of atoms. We already noted in Sect. 2 that it would be a good idea to allow the declaration of distinct sorts of atoms (for example, to more easily encode distinct sorts of names in an object language). Indeed, it might be a good idea to allow atom polymorphism via Haskell-style type classes [15], with a type class for 'types of atoms'. But even with a single sort of atoms, there is good reason to consider notions of abstraction in which the data to the left of the 'abstraction dot' is structurally more complicated than just single atoms. For example, some object languages use operators that bind varying numbers of variables rather than having a fixed 'arity' (for example, the free identifiers in an ML match m, however many there may be, become bound in the function expression `fn`$\{m\}$). To encode such operators we can make use of the following FreshML data type construction

```
datatype 'a abs = Val of 'a | Abs of [atm]('a abs)
```

whose values `Abs a1.(Abs a2.`$\cdots$`Val` *val*) are some finite number of atom-abstractions of a value *val* of type `'a`. When specifying a function on `'a abs` by structural recursion, one has to recurse on the list of binders in such a value, whereas in practice one usually wants to recurse directly on the structure of the inner most value *val*. Therefore it would be useful to have 'atom-list abstraction types' `[atm list]`*ty* (denotationally isomorphic to *ty* `abs`) and abstraction expressions of the form *as*`.`*exp*, where *as* is a value of type `atm list` and *exp* is an expression of type *ty*. But if one abstracts over atom-lists, why not over other concrete types built up from atoms? Indeed, in addition to such concrete types, it appears to be useful to abstract with respect to certain abstract types, such as finite sets of atoms, or finite mappings defined on atoms. So it may be appropriate to consider a general form of abstraction type, `[`*ty*`]`ty', for arbitrary types *ty* and ty'. To do so would require some changes to the nature of the freshness judgement (7), whose details have yet to be worked out. In fact, the FM-sets model contains a set-former for such a general notion of abstraction, so there is a firm semantic base from which to explore this extension of FreshML. Emphasising this firm semantic base is a good note on which to finish. Having reached this far, we hope the reader agrees that FreshML has some novel and potentially useful features for metaprogramming modulo renaming of bound variables. But whatever the particularities of FreshML, we believe the real source of this potential is the FM-sets model, which appears to provide a simple, elegant and useful mathematical foundation for computing and reasoning about name binding modulo renaming. It is certainly the case that without it, we would not have reached the language design described in this paper. (So please read the Appendix!)

A Appendix: FM-Sets

Naïvely speaking, ML types and expressions denote sets and functions. By contrast, FreshML types and expressions are intended to denote *FM-sets* and *equivariant functions*. This appendix gives a brief review these notions; more details can be found in [7]. Of course, the presence of recursive features and computational effects in ML means that a denotational semantics for it really involves much more complicated mathematical structures than mere sets. Similarly, to account for the recursive features of the present version of FreshML, we should really give it a denotational semantics using domains and continuous functions in the category of FM-sets. For simplicity's sake we suppress these domain-theoretic details here.

Notation. Let $\mathbb{A} = \{a, a', \ldots\}$ be a fixed countably infinite set, whose elements we call *semantic atoms*. Let $S_{\mathbb{A}}$ denote the group of all permutations of $\mathbb{A}$. Thus the elements π of $S_{\mathbb{A}}$ are bijections from $\mathbb{A}$ to itself. The group multiplication takes two such bijections π and π' and composes them—we write the composition of π followed by π' as $\pi'\pi$. The group identity is the identity function on $\mathbb{A}$, denoted by $id_{\mathbb{A}}$.

Recall that an *action* of the group $S_{\mathbb{A}}$ on a set X is a function $(-) \cdot_X (-)$ mapping pairs $(\pi, x) \in S_{\mathbb{A}} \times X$ to elements $\pi \cdot_X x \in X$ and satisfying

$$\pi' \cdot_X (\pi \cdot_X x) = (\pi'\pi) \cdot_X x \quad \text{and} \quad id_{\mathbb{A}} \cdot_X x = x$$

for all $\pi, \pi' \in S_{\mathbb{A}}$ and $x \in X$. For example, if Λ is the set of syntax trees of lambda terms with variable symbols from $\mathbb{A}$

$$\Lambda = \{t ::= a \mid t\,t \mid \lambda a.t \ (a \in \mathbb{A})\} \tag{38}$$

then there is a natural action of $S_{\mathbb{A}}$ on Λ: for each $\pi \in S_{\mathbb{A}}$ and $t \in \Lambda$, $\pi \cdot_{\Lambda} t$ is the tree which results from permuting all the atoms occurring in t according to π.

In general, an action of $S_{\mathbb{A}}$ on a set X gives us an abstract way of regarding the elements x of X as somehow 'involving atoms from $\mathbb{A}$ in their construction', in as much as the action tells us how permuting atoms changes x—which turns out to be all we need for an abstract theory of renaming and binding. An important part of this theory is the notion of *finite support.* This generalises the property of an abstract syntax tree that it only involves finitely many atoms in its construction to the abstract level of an element of any set equipped with an $S_{\mathbb{A}}$-action.

Definition A.1 (Finite support). Given a set X equipped with an action of $S_{\mathbb{A}}$, a set of semantic atoms $\omega \subseteq \mathbb{A}$ is said to *support* an element $x \in X$ if all permutations $\pi \in S_{\mathbb{A}}$ which fix every element of ω also fix x:

$$(\forall a \in \omega \,.\, \pi(a) = a) \Rightarrow \pi \cdot_X x = x \ . \tag{39}$$

We say that x is *finitely supported* if there is some finite subset $\omega \subseteq \mathbb{A}$ supporting it. It is not too hard to show that if x is finitely supported, then there is a smallest finite subset of $\mathbb{A}$ supporting it: we call this the *support* of x, and denote it by $\mathit{supp}_X(x)$.

Definition A.2 (The category FM-Set). An *FM-set* is a set X equipped with an action of the permutation group $S_{\mathbb{A}}$ in which every element $x \in X$ is finitely supported. These are the objects of a category, **FM-Set**, whose morphisms $f : X \longrightarrow Y$ are *equivariant functions*, i.e. functions from X to Y satisfying

$$f(\pi \cdot_X x) = \pi \cdot_Y f(x)$$

for all $\pi \in S_{\mathbb{A}}$ and $x \in X$.

Example A.3. The set Λ of untyped lambda terms, defined as in (38) and with the $S_{\mathbb{A}}$-action mentioned there, is an FM-set. The support of $t \in \Lambda$ is just the finite set of *all* variable symbols occurring in the tree t (whether free, bound, or binding). Note that if two lambda terms are α-equivalent, $t =_\alpha t'$, then for any permutation π one also has $\pi \cdot_\Lambda t =_\alpha \pi \cdot_\Lambda t'$. It follows that $(-) \cdot_\Lambda (-)$ induces an action on the quotient set $\Lambda/{=_\alpha}$. It is not hard to see that this is also an FM-set, with the support of an α-equivalence class of lambda terms being the finite set of *free* variable symbols in any representative of the class (it does not matter which). This turns out to be the denotation of the data type `lam` declared in Fig. 1 (using [7, Theorem 5.1]).

It should be emphasised that the above definitions are not novel, although the use to which we put them is. They are part of the rich mathematical theory of continuous actions of topological groups. $S_{\mathbb{A}}$ has a natural topology as a subspace of the countably infinite product of the discrete space $\mathbb{A}$, which makes it a topological group. Given an action of $S_{\mathbb{A}}$ on a set X, all the elements of X have finite support if and only if the action is a continuous function $S_{\mathbb{A}} \times X \longrightarrow X$ when X is given the discrete topology. Thus **FM-Set** is an example of a category of 'continuous G-sets' and as such, much is known about its properties: see [7, Sect. 6] for references. Here we just recall its cartesian closed structure.

A.1 Products in FM-Set

These are given by taking the cartesian product of underlying sets

$$X \times Y = \{\, (x,y) \mid x \in X \text{ and } y \in Y \,\}\,.$$

The permutation action is given componentwise by that of X and Y:

$$\pi \cdot_{X \times Y} (x,y) \triangleq (\pi \cdot_X x, \pi \cdot_Y y) \qquad (\pi \in S_{\mathbb{A}}).$$

Each pair $(x,y) \in X \times Y$ does indeed have finite support, namely $\mathit{supp}_X(x) \cup \mathit{supp}_Y(y)$.

A.2 Exponentials in FM-Set

Given any function (not necessarily an equivariant one) $f : X \longrightarrow Y$ between FM-sets X and Y, we can make permutations of $\mathbb{A}$ act on f by defining

$$\pi \cdot_{X \to Y} f \triangleq \lambda x \in X.(\pi \cdot_Y f(\pi^{-1} \cdot_X x)) \ . \tag{40}$$

By applying the property (39) with $\cdot_{X \to Y}$ in place of $\cdot_X$, it makes sense to ask whether f has finite support. The subset of all functions from X to Y which do have finite support in this sense, together with the action $\cdot_{X \to Y}$ given by (40), forms an FM-set which is the exponential $X \to Y$ in **FM-Set**. Note that (40) implies that for all $\pi \in S_{\mathbb{A}}$, $f \in (X \to Y)$ and $x \in X$

$$\pi \cdot_Y f(x) = (\pi \cdot_{X \to Y} f)(\pi \cdot_X x)$$

and hence evaluation $(f,x) \mapsto f(x)$ determines an equivariant function $\mathit{ev} : (X \to Y) \times X \longrightarrow Y$. Given any $f : Z \times X \longrightarrow Y$ in **FM-Set**, its exponential transpose $\mathit{cur}(f) : Z \longrightarrow (X \to Y)$ is given by the usual 'curried' version of f, for the following reason: given any $z \in Z$, if $\pi \in S_{\mathbb{A}}$ fixes each semantic atom in $\mathit{supp}_Z(z)$, then it is not hard to see from definition (40) and the equivariance of f that π fixes the function $\lambda x \in X.f(z,x)$ as well; so this function has finite support and hence is in $X \to Y$. So defining $\mathit{cur}(f)(z)$ to be $\lambda x \in X.f(z,x)$, we get a function $\mathit{cur}(f) : Z \longrightarrow (X \to Y)$; it is equivariant because f is and clearly has the property required for it to be the exponential transpose of f.

In the rest of this appendix we indicate the structures in the category **FM-Set** used to model the key novelties of FreshML: locally fresh atoms, atom abstraction and concretion of atom abstractions.

A.3 Locally fresh atoms

The set $\mathbb{A}$ of atoms becomes an object of **FM-Set** once we endow it with the action:

$$\pi \cdot_{\mathbb{A}} a \triangleq \pi(a) \quad (\pi \in S_{\mathbb{A}}, a \in \mathbb{A}).$$

(The support of $a \in \mathbb{A}$ is just $\{a\}$.) This FM-set is the denotation of the type `atm` of atoms in FreshML.

The meaning of `new a in` *exp* `end` in **FM-Set** is given by the following proposition (whose straightforward proof we omit). It makes use of the following construction in the category **FM-Set**: given an FM-set G, define

$$\mathbb{A} \otimes G \triangleq \{\ (a, g) \in \mathbb{A} \times G \mid\ a \notin \mathit{supp}_G(g)\ \}\quad .$$

This becomes an FM-set if we define a permutation action by

$$\pi \cdot_{\mathbb{A}\otimes G} (a, g) \triangleq (\pi(a), \pi \cdot_G g)\quad .$$

Proposition A.1. *Given a morphism* $e : \mathbb{A} \otimes G \longrightarrow T$ *in* **FM-Set** *satisfying*

$$a \notin \mathit{supp}_T(e(a, g)) \quad \text{for all } (a, g) \in \mathbb{A} \otimes G \tag{41}$$

then there is a unique morphism $\mathit{new}(e) : G \longrightarrow T$ *such that*

$$\mathit{new}(e)(g) = e(a, g) \quad \text{for all } g \in G \text{ and } a \notin \mathit{supp}_G(g) \tag{42}$$

is satisfied. □

If a typing context Γ has denotation G and if $\mathtt{a} \notin \mathrm{dom}(\Gamma)$, then the denotation of the typing context $(\mathtt{a} : \mathtt{atm}) \otimes \Gamma$ used in rule (11) of Fig. 2 is the FM-set $\mathbb{A} \otimes G$. Suppose the denotation of $(\mathtt{a} : \mathtt{atm}) \otimes \Gamma \vdash \mathit{exp} : \mathit{ty}$ is the equivariant function $e : \mathbb{A} \otimes G \longrightarrow T$. We can use Proposition A.1 to give a denotation to $\Gamma \vdash (\mathtt{new\ a\ in}\ \mathit{exp}\ \mathtt{end}) : \mathit{ty}$ as $\mathit{new}(e) : G \longrightarrow T$, using the fact that the freshness part of the hypothesis of (11) means that e satisfies condition (41). The soundness of rule (11) follows from the defining property (42) of $\mathit{new}(e)$.

A.4 Abstraction and Concretion

If the denotation of the FreshML type *ty* is the FM-set T, the denotation of the atom abstraction type `[atm]`*ty* is the FM-set $[\mathbb{A}]T$ introduced in Sect. 4 of [7]. Its underlying set is the quotient of the cartesian product $\mathbb{A} \times T$ by the equivalence relation $\sim_{\mathbb{A}}$ defined as follows: $(a, t) \sim_{\mathbb{A}} (a', t')$ holds if and only if

$$(a''\ a) \cdot_T t = (a''\ a') \cdot_T t'$$

holds for some (or equivalently, any) a'' not in the support of a, a', t, or t' (where $(a''\ a) \in S_{\mathbb{A}}$ denotes the permutation interchanging a'' and a). We write $a.t$ for the $\sim_{\mathbb{A}}$-equivalence class determined by (a,t).[7] The action of a permutation $\pi \in S_{\mathbb{A}}$ on elements of $[\mathbb{A}]T$ is given by:

$$\pi \cdot_{[\mathbb{A}]T} (a.t) \triangleq (\pi(a)).(\pi \cdot_T t) \ .$$

This does indeed determine a well-defined FM-set, with the support of $a.t$ being the finite set $\mathit{supp}_T(t) \setminus \{a\}$.

If `a` is associated with the semantic atom $a \in \mathbb{A}$ (in some given value environment) and the denotation of $\mathit{exp} : \mathit{ty}$ is $t \in T$, then the denotation of the atom abstraction expression `a.`*exp* is $a.t \in [\mathbb{A}]T$.

On the other hand, the meaning of concretion expressions (Sect. 4) uses the following property of FM-sets.

Proposition A.2. *Given an FM-set T, for each atom-abstraction $e \in [\mathbb{A}]T$ and semantic atom $a \in \mathbb{A}$, if $a \notin \mathit{supp}_{[\mathbb{A}]T}(e)$ then there is a unique element $e@a$ of T such that $e = a.(e@a)$.*

When $e = a'.t$ then $a \notin \mathit{supp}_{[\mathbb{A}]T}(e)$ if and only if $a = a'$ or $a \notin \mathit{supp}_T(t)$; and in this case $(a',t) \sim_{\mathbb{A}} (a, (a\ a') \cdot_T t)$. So when $a = a'$ or $a \notin \mathit{supp}_T(t)$, it follows from the uniqueness part of the defining property of $e@a$ that

$$(a'.t)@a = (a\ a') \cdot_T t \tag{43}$$

holds. □

The partial function $e, a \mapsto e@a$ is used to give the denotational semantics of concretion expressions, *exp*`@a`. The soundness of rule (13) follows from the easily verified fact that $\mathit{supp}_T(e@a) \subseteq \mathit{supp}_{[\mathbb{A}]T}(e) \cup \{a\}$.

Acknowledgements

The authors thank members of the Cambridge University *Logic and Semantics* group, Mark Shinwell (who is carrying out an implementation of FreshML), Simon Peyton Jones and the anonymous referees for constructive comments on the work presented here. This research was partially supported by the ESPRIT Working Group Nr 26142 on Applied Semantics (APPSEM).

References

[1] R. Burstall, D. MacQueen, and D. Sannella. HOPE: An experimental applicative language. In *Proc. LISP Conference, Stanford CA, 1980*, pages 136–143. Stanford University, 1980.

[2] R. M. Burstall. Design considerations for a functional programming language. In *Proc. of the Infotech State of the Art Conference, Copenhagen*, 1977.

[3] G. Cousineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[7] The notation $[a]t$ was used for this in [7].

[4] 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.

[5] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, Washington, 1999.

[6] M. J. Gabbay. *A Theory of Inductive Definitions with α-Conversion: Semantics, Implementation, and Meta-Language*. PhD thesis, Cambridge University, in preparation.

[7] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[9] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, Washington, 1999.

[10] D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, 1990.

[11] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[12] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[13] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10:470–502, 1988.

[14] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.

[15] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language*. February 1999. Available from `<http:www.haskell.org>`.

[16] 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.