Technical Report

UCAM-CL-TR-621
ISSN 1476-2986

Number 621

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Fresh Objective Caml user manual

Mark R. Shinwell, Andrew M. Pitts

February 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Contents

1 Introduction **5**

1.1 About the language 5

1.2 Obtaining and installing Fresh O'Caml 5

1.3 The toplevel system 6

1.4 Batch compilation 6

2 Concrete syntax **9**

2.1 Keywords 9

2.2 Values 9

2.3 Type expressions 10

2.4 Patterns 10

2.5 Expressions 10

3 Handling of binding operations **11**

3.1 Types of bindable names 11

3.2 Abstraction values and types 11

3.3 Declaring fresh atoms 12

3.4 Structural equality and alpha-conversion 13

3.5 Abstraction patterns 14

3.6 Swapping bindable names 16

3.7 The freshness relation 18

3.8 Abstraction types in general 18

3.9 Reference cells 20

Bibliography **21**

1 Introduction

1.1 About the language

Fresh O'Caml extends Objective Caml (`www.ocaml.org`) to provide seamless and correct manipulation of object language binding constructs inside a metalanguage. The language enables the programmer to concentrate on the algorithmic essence of their syntax-manipulating programs, without having to concern themselves with the tedious matters of ensuring that properties of α-convertible object language names are correctly respected. Importantly, an algebraic datatype in Fresh O'Caml may involve the use of a novel *abstraction type*-former, which enables one to incorporate representations of object-level binding operations into meta-level syntax trees without any further ado. Such representations are first-class citizens in the metalanguage just like products, sums and the like; the resulting syntax trees correctly represent object-level syntax up to α-conversion of the bound names, as proved in [4]. The result is a style of metaprogramming which feels natural and elegant, yielding concise name- and binder-manipulating code which exhibits real correspondence to the underlying algorithms. Our approach differs from approaches which use higher-order abstract syntax internally, not least because Fresh O'Caml gives direct access to the names of bound entities.

Fresh O'Caml is the latest in a series of languages based on the Gabbay-Pitts theory of FM-sets and metaprogramming [1]. This theory first yielded a language with quite a complicated static type system for controlling the dynamic effects of fresh name generation [3]. More recently, such a type system has been found to be unnecessary for the basic correctness properties of representing object-level syntax up to α-conversion of bound names; as a result of this discovery, the FreshML language described in [6] was created. Whilst an interpretive system was created for this language, it is no longer supported. Instead, Fresh O'Caml is now used, which transfers these ideas to the Objective Caml language.

This *User Manual* provides an informal introduction to the features of Fresh O'Caml that are new compared with O'Caml. It assumes familiarity with the ML programming idiom and terminology in general and with O'Caml [2] in particular. A very much more detailed discussion of the Fresh O'Caml language and its correctness properties are provided in a journal paper [5] and Shinwell's PhD thesis [4].

1.2 Obtaining and installing Fresh O'Caml

To obtain Fresh O'Caml, visit the website at `www.fresh-ocaml.org`, which also contains examples of programming with binders using it. The Fresh O'Caml code samples referred to in this manual are linked to from that page.

The current release of Fresh O'Caml is based on release 3.08.2 of the Objective Caml system. It is known to compile successfully on Linux and Mac OS X systems and should work without difficulty on other UNIX-like platforms. It has not been tested on Microsoft Windows.

Fresh O'Caml is currently only distributed as source code. You can download the latest distribution, which at the time of writing is version 3.08.2+4 of 14th January 2005, or browse the example files. Commands similar to the following should suffice to build and install the system:

```
tar fxz fresh-ocaml-3.08.2+4.tar.gz
cd fresh-ocaml-3.08.2+4
./configure
make world
make opt
make install
```

If you wish to install the system somewhere other than the default installation location of `/usr/local/`, then use the `-prefix` option to the `configure` script, for example:

```
./configure -prefix ~/fresh-ocaml
```

Note that **we strongly advise not installing Fresh O'Caml with the same prefix as your existing O'Caml installation**.

The web site `www.fresh-ocaml.org` will be updated with revised Fresh O'Caml distributions and/or installation instructions as they appear.

1.3 The toplevel system

The interpreter is started by executing `fresh-ocaml`. This produces a modified version of the O'Caml interactive toplevel system waiting for you to type Fresh O'Caml programs.

```
$ fresh-ocaml
        Fresh Objective Caml version 3.08.2+4

#
```

Write your programs using the concrete syntax of O'Caml [2, Chapter 6] augmented by the syntax described in Chapter 2. The features that Fresh O'Caml adds to O'Caml are described in Chapter 3.

1.4 Batch compilation

For batch compilation, the following compilers are provided. These correspond to those from the standard Objective Caml distribution. (The features that Fresh O'Caml adds to O'Caml are described in Chapter 3.)

- `fresh-ocamlc`: compilation to bytecode.

- `fresh-ocamlc.opt`: compilation to bytecode, but with the compiler itself compiled to native code.
- `fresh-ocamlopt`: compilation to native code.
- `fresh-ocamlopt.opt`: compilation to native code, but with the compiler itself also compiled to native code.

2 Concrete syntax

This chapter describes the additions Fresh O'Caml makes to the concrete syntax of O'Caml. The meaning of the new syntactic forms is described in Chapter 3.

2.1 Keywords

The following identifiers are reserved as keywords in Fresh O'Caml and cannot be employed otherwise:

```
name   fresh   freshfor   swap
```

The following character sequences are also keywords:

```
<<   >>   <|   |>
```

2.2 Values

Atoms

There is a (countably infinite) collection of *atoms*.[1] Atoms form a name-space distinct from other O'Caml name-spaces, such as names of record fields or methods. (They are the values inhabiting the types of bindable names, `'a name`; see section 3.1.)

Abstraction values

Abstraction values are given by pairs, written $[v_1]v_2$, where v_1 and v_2 are values. They are the values inhabiting abstraction types, which are written << $tyexpr_1$ >> $tyexpr_2$: see section 3.2.

[1] The name 'atom' is used for historical reasons to do with the mathematical universe [1] that underlies the semantics of Fresh O'Caml.

2.3 Type expressions

Fresh O'Caml adds to O'Caml a distinguished type constructor for *types of bindable names* and a binary operation for forming *abstraction types*.

typexpr	::=	*tyexpr* `name`	type of names
	|	`<<` *tyexpr* `>>` *tyexpr*	abstraction type
	···	rest of grammar as in [2, Sect. 6.4]	

2.4 Patterns

Fresh O'Caml adds to O'Caml patterns for abstraction values (see section 3.5). There are two alternative forms of syntax. Since the first one (`<< >>`) interferes with the Camlp4 preprocessor quotation mechanism, it is recommended to use the second one (`<| |>`) if Camlp4 is in use.

pattern	::=	`<<` *pattern* `>>` *pattern*	abstraction,
	|	`<|` *pattern* `|>` *pattern*	with alternative syntax
	···	the rest as in [2, Sect. 6.6]	

2.5 Expressions

Fresh O'Caml adds to O'Caml expressions for creating fresh bindable names (see section 3.3), for swapping bindable names (see section 3.6), for abstractions (see section 3.2), and for testing whether a bindable name is fresh for an expression (see section 3.7).

expr	::=	`fresh`	fresh bindable name
	|	`swap` *expr* `and` *expr* `in` *expr*	swapping
	|	`<<` *expr* `>>` *expr*	abstraction,
	|	`<|` *expr* `|>` *expr*	with alternative syntax
	|	*expr* `freshfor` *expr*	freshness relation
	···	rest of grammar as in [2, Sect. 6.7]	

The typing of these expressions can be deduced from the following interaction with the Fresh O'Caml toplevel. (As can be seen from the first response, the interactive system prints atoms as `name_`n, for $n = 0, 1, 2, \ldots$)

```
# fresh;;
- : 'a name = name_0
# function x -> function y -> function z -> swap x and y in z;;
- : 'a name -> 'a name -> 'b -> 'b = <fun>
# function x -> function y -> << x >> y;;
- : 'a -> 'b -> <<'a>>'b = <fun>
# function x -> function y -> <| x |> y;;
- : 'a -> 'b -> <<'a>>'b = <fun>
# function x -> function y -> x freshfor y;;
- : 'a name -> 'b -> bool = <fun>
```

3 Handling of binding operations

This chapter gives an informal explanation of Fresh O'Caml's facilities for computing with binders.

3.1 Types of bindable names

Fresh O'Caml provides a polymorphic family `'a name` of types of *bindable names*. For each type $tyexpr$, the values of type $tyexpr$ `name` are called *atoms* and behave rather like values of the ML type `unit ref`; they provide an inexhaustible supply of names which may take part in binding operations at the object level. The user has access to this supply via the expression `fresh` of type `'a name`, which yields a fresh atom each time it is evaluated. Each type of bindable names is isomorphic to any other; the use of the type-parameterised family `'a name` is simply to provide as many different such types as the user needs, whilst subsuming polymorphism over the different kinds of name under the usual mechanisms for polymorphic types.[1]

3.2 Abstraction values and types

We are going to use as a running example of an object language a small ML-like language with the following kinds of terms:

x	variables
$fn\ x \Rightarrow e$	function abstraction
$e_1\, e_2$	function application
$let\ fun\ f\ x = e_1\ in\ e_2\ end$	local recursive function

We will name variables (such as x and f in the above terms) using a type of bindable names, in order to be able to take advantage of Fresh O'Caml's ability to handle α-conversion and freshness of atoms automatically:

```
# type t and var = t name;;
```

to which the system responds

```
type t
type var = t name
```

[1] This is a simpler approach than the `bindable_type` declaration used by FreshML [6].

It is the type `var` that we want to use as a type of bindable names; the abstract type `t` just provides us with some type (a fresh one, because of the generative nature of the declaration `type t` in [Fresh] O'Caml) to which to apply the type construction `name`.

Now we can declare a suitable datatype for parse trees of terms in our example object-language, or rather, for α-equivalence classes of them:

```
# type term =                          (* terms, e *)
    Var of var                         (* x *)
  | Fn of <<var>>term                  (* fn x => e *)
  | App of term * term                 (* e1 e2 *)
  | Letfun of                          (* let fun f x = e1 in e2 *)
      <<var>>((<<var>>term) * term);;
```

The types we have used in this declaration tell the system exactly which data constructors are binders and in which way their arguments are bound. For example in the term $let\ fun\ f\ x = e_1\ in\ e_2\ end$, there is a binding occurrence of the variable f whose scope is both of e_1 and e_2; and a binding occurrence of the variable x whose scope is just e_1. These binding scopes are reflected by the type of corresponding constructor `Letfun`. For example, if a and b are atoms of type `var`, then

$$[a]([b](\texttt{App}(\texttt{Var}\,a, \texttt{Var}\,b)), \texttt{Var}\,a)$$

is a value of type `<<var>>((<<var>>term)*term)`; and then

$$\texttt{Letfun}([a]([b](\texttt{App}(\texttt{Var}\,a, \texttt{Var}\,b)), \texttt{Var}\,a)) \tag{3.1}$$

is a value of type `term` representing the α-equivalence class of the term $let\ fun\ f\ x = f\ x\ in\ f\ end$. For example, renaming a to some fresh atom a', then the value

$$\texttt{Letfun}([a']([b](\texttt{App}(\texttt{Var}\,a', \texttt{Var}\,b)), \texttt{Var}\,a'))$$

behaves in all respects like (3.1). Fresh O'Caml codifies a common practice in mathematics when it comes to dealing with binders: its programs allow the user to manipulate α-equivalence classes by naming one of their representatives, through the use of value identifiers naming specific bindable names. However, as shown in [5], Fresh O'Caml is designed in such a way that well-typed programs can only use such representatives in ways that are independent up α-equivalence of which particular representative is chosen.

3.3 Declaring fresh atoms

We can create values of type `term` by applying its constructors to previously constructed values, with the constructor `Var` getting us off the ground. However, initially there are no constants of type `var` and to get some bindable names we have to create them using the `fresh` expression. For example, to get a parse tree of type `term` that represents the (α-equivalence class of the) first projection function $fn\ x \Rightarrow fn\ y \Rightarrow x$, we can use the declarations

```
let x,y = (fresh:var),(fresh:var);;
let fst = Fn(<<x>>(Fn(<<y>>(Var x))));;
```

The first one associates with the value identifiers `x` and `y` atoms not used so far in the current environment, a and b say; and then the second one associates with the value identifier `fst` the value `Fn`$([a](\texttt{Fn}([b](\texttt{Var}\,a))))$ of type `term`.

Note that `<< x >>` $(-)$ *is not a meta-level binding operation*. Thus in the above declaration of `fst`, the occurrences of `x` and `y` are not locally bound value identifiers (as they would be in `function x -> function y -> Var x`, for example), but refer to the previously declared values of type `var`. For example, we can re-use those values to declare an open term:

```
# let e0 =  Fn(<<y>>(Var x));;
```

The system's response to the above declarations is:

```
val x : var = name_0
val y : var = name_0
val fst : term = Fn <<name_0>>(Fn <<name_1>>(Var name_0))
val e0 : term = Fn <<name_1>>(Var name_0)
```

This tells us that we have associated with `fst` and `e0` values of type `term` and it gives us a textual representation of those values. In this representation atoms are printed as `name_`n, with distinct atoms getting distinct numbers n. Note from the first two lines of the system's response that this numbering works on a 'per-response' basis (rather like the renaming of type variables by an ML system to start from `'a` for each response): the particular atoms assigned to each name are hidden from the user.

3.4 Structural equality and alpha-conversion

The important correctness property of Fresh O'Caml is that *values of types like* `term` *are observationally equivalent iff they correspond to* α*-equivalent terms of the object-language*.[2] Thus we obtain working 'up to α-conversion' for free. In particular, when the function

```
(=) : 'a -> 'a -> bool
```

that tests for *structural equality* [2, Sect. 19.2] is applied to values of type `term`, it computes object-level α-equivalence. For example consider the following declarations:

```
let e1 = Fn(<<x>>(Fn(<<x>>(App(Var x, Var x)))));;
let e2 = Fn(<<x>>(Fn(<<y>>(App(Var x, Var x)))));;
let e3 = Fn(<<x>>(Fn(<<y>>(App(Var y, Var y)))));;
```

These associate with the identifiers `e1`, `e2` and `e3` values of type `term` corresponding to the α-equivalence classes of the terms

$$fn\ x \Rightarrow fn\ x \Rightarrow x\,x$$
$$fn\ x \Rightarrow fn\ y \Rightarrow x\,x$$
$$fn\ x \Rightarrow fn\ y \Rightarrow y\,y$$

respectively. The first and the second of these terms are not α-equivalent, whereas the first and the third are. Fresh O'Caml knows this:

[2] This is discussed at length in [4].

```
# e1 = e2;;
- : bool = false
# e1 = e3;;
- : bool = true
```

3.5 Abstraction patterns

Deconstruction of an abstraction value $[v_1]v_2$ is performed using an abstraction pattern << *pattern*$_1$ >> *pattern*$_2$. This matches against the structure of the values in the binding position and in the body of the abstraction, rather like a pair pattern. Crucially however, during matching Fresh O'Caml ensures that any atoms which occur inside these values and which are bound by an abstraction are suitably freshened before the values are associated with identifiers in the environment. This is achieved by consistently *swapping* these existing atoms inside the values with as many fresh ones as necessary. This form of matching ensures that these concrete values are always 'suitably fresh'; many side-conditions to do with the freshness of names which crop up in operational semantics and the like are then automatically satisfied.

For example, consider the following function

```
subst : term -> var -> term -> term.
```

It implements *capture-avoiding* substitution: `subst e x e'` computes (a representation of) the object-language term obtained by capture-avoiding substitution of the term represented by `e` for all free occurrences of the variable named `x` in the term represented by `e'`.

```
# let rec subst e x e' =
    match e' with
        Var y -> if x = y then e else Var y
      | Fn(<<y>>e1) -> Fn(<<y>>(subst e x e1))
      | App(e1,e2) -> App(subst e x e1, subst e x e2)
      | Letfun(<<f>>(<<y>>e1,e2)) ->
          Letfun(<<f>>(<<y>>(subst e x e1), subst e x e2));;
```

For example, with `e0` as declared above,

```
# let e4 = subst (Var y) x e0;;
val e4 : term = Fn <<name_1>>(Var name_0)
# let e5 = Fn(<<x>>(Var y));;
val e5 : term = Fn <<name_1>>(Var name_0)
# e4 = e5;;
- : bool = true
```

The value binding for `e4` carries out the substitution of y for the free occurrence of x in the term represented by `e0`, which is $fn\ y \Rightarrow x$. We expect to obtain as a result a value corresponding not to the term $fn\ y \Rightarrow y$, but rather to the term $fn\ z \Rightarrow y$, where z is any variable other than y (such as x). As the above interaction indicates, this is indeed what we get.

The declaration of `subst` manages to implement this capture-avoiding behaviour because of the way Fresh O'Caml matches atom-abstraction patterns using fresh atoms for abstracted identifiers. For example, when evaluating `subst (Var y) x e0`, we fall through to the second `match`-clause

```
      | Fn(<<y>>e1) -> Fn(<<y>>(subst e x e1))
```

and reach a point where the environment contains the associations `x` $\mapsto a$, `e` $\mapsto$ (`Var` b) (where b is the atom the environment associates with `y`) and we are attempting to match `Fn(<<y>>e1)` against the value associated with `e0`, namely `Fn`$([b](\texttt{Var}\, a))$. The match succeeds in associating `y` with a fresh atom, b' say, and `e1` with the result of swapping b with b' in `Var` a, which is `Var` a again. Then the right-hand side of the clause, `Fn(<<y>>(subst x e e1))`, is evaluated in this environment (and with the current state augmented by b'), resulting in the value `Fn`$([b'](\texttt{Var}\, b))$, as expected.

So: *matching that involves abstraction patterns may have the side-effect of dynamically generating fresh names*.

For example consider the following declaration:

```
# let rec listBvars e =
    match e with
        Var _ -> []
      | Fn(<<x>>e1) -> x :: (listBvars e1)
      | App(e1,e2) -> (listBvars e1) @ (listBvars e2)
      | Letfun(<<f>>(<<x>>e1,e2)) ->
          f :: x :: (listBvars e1) @ (listBvars e2);;
val listBvars : term -> var list = <fun>
```

Perhaps the intention was for `(listBvars e)` to return a list of names of binding variables in the object-level term represented by `e`. However, such a list of binding variables makes no sense at the level of α-equivalence classes of object-level terms, i.e. values of type `term`; and indeed all `listBvars` does is to return a list of *fresh* atoms, rather than the actual atoms occurring in some value to which it is applied:

```
# listBvars e1;;
- : var list = [name_0; name_1]
```

We can count the length of such a list, but we do not have access to the names of the atoms in it. Compare this with the following declaration of a function `listFvars` (which makes use of the `List.filter` function [2, Sect. 20.17]):

```
# let rec listFvars e =
    match e with
        Var x -> [x]
      | Fn(<<x>>e1) -> List.filter ((<>) x) (listFvars e1)
      | App(e1,e2) -> (listFvars e1) @ (listFvars e2)
      | Letfun(<<f>>(<<x>>e1,e2)) ->
          List.filter ((<>) f) (
            (List.filter ((<>) x) (listFvars e1))
            @ (listFvars e2));;
val listFvars : term -> var list = <fun>
```

When applied to a value of type `term`, the function `listFvars` returns a list of atoms (possibly with repeats) corresponding to the free variables of the object-language term represented by the value. Although we cannot see the names of those atoms directly, we can compute with them. The following interaction illustrates the difference between `listBvars` and `listFvars` (recall that `e0` was declared to be `Fn(<<y>>(Var x))`):

```
# (listFvars e0) = [x];;
- : bool = true
# (listBvars e0) = [y];;
- : bool = false
```

The second equality is `false` because `(listBvars e0)` computes a fresh atom for `e0`'s bound variable. An even more extreme example is:

```
# (listBvars e0) = (listBvars e0);;
- : bool = false
```

The value is `false` because `(listBvars e0)` computes a fresh atom each time it is evaluated.

3.6 Swapping bindable names

Fresh O'Caml has name-swapping expressions of the form

$$\texttt{swap}\ expr_1\ \texttt{and}\ expr_2\ \texttt{in}\ expr_3$$

Here $expr_1$ and $expr_2$ must be expressions of the same type of bindable names. The `swap`-expression evaluates $expr_1$ and $expr_2$ to find which atoms they stand for, and then interchanges all occurrences of those atoms in the value obtained by evaluating $expr_3$. For example:

```
# (swap x and y in App(Var x, Var y)) = App(Var y, Var x);;
- : bool = true
```

Name-swapping can express the operation from [3] of *concreting* an abstraction at a (fresh) atom:

```
# let (@@) =
    function <<x>>(e:term) ->
    function (x':var) ->
      swap x and x' in e;;
val ( @@ ) : <<var>>term -> var -> term = <fun>
# (<<x>>(Fn(<<y>>(App(Var x, Var x))))) @@ y;;
- : term = Fn <<name_1>>(App (Var name_0, Var name_0))
```

A more typical use of `swap`-expressions is to replace the bound names in a number of abstraction with the same name. For example, consider declaring a function `eq` of type `term -> term -> bool` to compute equality of α-equivalence classes of terms in the fragment of ML we are using as a running example. When comparing the

equivalence classes of two function abstractions we can always choose representatives of the classes that have the same fn-bound variable and then compare the bodies of the function abstractions for equality modulo α-equivalence. Thus we might be tempted to use a clause like

```
      | (Fn(<<x>>e1), Fn(<<x>>e2)) -> eq e1 e2
```

in the declaration of `eq`. This is not possible in Fresh O'Caml, because for simplicity it adopts the same linearity constraint on patterns as does O'Caml: *there must be at most one occurrence of any value identifier in a pattern*. In the clause for `Fn` in the declaration of `eq`, we must use distinct bound names in the function abstractions and then swap them; and similarly for the other binder, `Letfun`:

```
# let rec eq e e' =
    match (e, e') with
        (Var x, Var y) -> (x = y)
      | (Fn(<<x1>>e1), Fn(<<x2>>e2)) ->
          eq (swap x1 and x2 in e1) e2
      | (App(e1,e1'), App(e2,e2')) ->
          (eq e1 e2) && (eq e1' e2')
      | (Letfun(<<f1>>(<<x1>>e1,e1')),
         Letfun(<<f2>>(<<x2>>e2,e2'))) ->
          (eq (swap f1 and f2 in (swap x1 and x2 in e1)) e2)
          && (eq (swap f1 and f2 in e1') e2')
      | (_, _) -> false;;
val eq : term -> term -> bool = <fun>
```

We mentioned in Sect. 3.4 that the built-in structural equality function `(=)` implements the appropriate notion of α-equivalence for values of type `term`; and in fact the function `eq` coincides with the built-in structural equality function for this type. For example, with `e1`, `e2` and `e3` as declared in Sect. 3.4:

```
#  eq e1 e2;;
- : bool = false
# eq e1  e3;;
- : bool = true
# let f = (fresh:var);;
val f : var = name_0
# let e6 = Letfun(<<f>>(<<x>>(Var x), App(Var f, Var y)));;
val e6 : expr =
  Letfun <<name_1>>(<<name_0>>(Var name_0),
  App (Var name_1, Var name_2))
# let e7 = Letfun(<<x>>(<<y>>(Var y), App(Var x, Var y)));;
val e7 : expr =
  Letfun <<name_1>>(<<name_0>>(Var name_0),
  App (Var name_1, Var name_2))
# eq e6 e7;;
- : bool = true
# e6 = e7;;
- : bool = true
```

Fresh O'Caml also provides a standard library module `Freshness` which contains a function to swap multiple atoms at once. Calling

```
Freshness.swap_multiple_atoms l1 l2 v
```

where `l1` and `l2` are lists of bindable names returns that value formed by swapping the first element of `l1` with the first element of `l2` throughout `v`, and so forth to the end of the lists. Both lists must therefore be the same length.

3.7 The freshness relation

The boolean-valued expression

$$expr_1 \texttt{ freshfor } expr_2$$

is true iff $expr_1$ is an expression of bindable name type that evaluates to an atom occurring in the *algebraic support* (which we here abbreviate to just *support*) of the value to which $expr_2$ evaluates. The notion of 'support' comes from the intended denotational semantics of Fresh O'Caml in the universe of FM-cppos: see [4]. It gives a syntax-independent notion of the set of free atoms of a value. We refer the reader to [1] for a full account of this notion and simply note that the support of a Fresh O'Caml representation of some object-language term corresponds to the set of free variables in that term. The `freshfor` relation thus provides a built-in 'not-a-free-variable-of' test for object-language terms. For example, the `freshfor` relation holds between an atom and a value of the type `term` declared in Sect. 3.2 just in case the atom does not occur in the list returned by applying the `listFvars` function of Sect. 3.5 to the value.

```
# x freshfor e0;;
- : bool = false
# List.mem x (listFvars e0);;
- : bool = true
# y freshfor e0;;
- : bool = true
# List.mem y (listFvars e0);;
- : bool = false
```

3.8 Abstraction types in general

So far we have given examples of abstraction types << $tyexpr_1$ >> $tyexpr_2$ and values $[v_1]v_2$ where $tyexpr_1$ is a type of bindable names and hence v_1 is an atom. Fresh O'Caml permits values of arbitrary type in binding position, that is, within the double angle brackets. Using some types of value in this position may cause `Invalid_argument` exceptions to be raised upon evaluation, in the same way that, for example, using structural equality between function values raises this exception [2, Sect. 19.2]. The reason for this has to do with the semantics of matching, alluded to in Sect. 3.5. Fresh O'Caml 'freshens' [ ]-bound values by swapping all the atoms in their support (see Sect. 3.7) with dynamically generated fresh ones. For this process to be semantically sound, the system has to calculate the support of a value accurately (overestimation by just finding all the atoms occurring in the value can be unsound). This is not possible for function values, so the `Invalid_argument` exception is raised upon encountering such values during matching. For example:

```
# let absApply <<f>>x = f x;;
val absApply : <<'a -> 'b>>'a -> 'b = <fun>
# absApply (<< function (x:int) -> x >> 0 );;
Exception: Invalid_argument "Invalid value in binding position".
```

The semantics of Fresh O'Caml matching ensures that values of the form $[v_1]v_2$ are indistinguishable up to freshening all the atoms in the support of the value v_1 which occur in the value v_2. For example, swapping atoms a and b with different atoms a' and b' respectively, the value

$$[\,[\,[a]b\,;\,[b]b\,]\,](a,b) \tag{3.2}$$

(of type `<<(<<'a name>>('a name)) list>>('a name * 'a name)`) behaves in all respects like

$$[\,[\,[a']b'\,;\,[b]b\,]\,](a,b') \tag{3.3}$$

but is a different value from

$$[\,[\,[a']b'\,;\,[b]b\,]\,](a',b') \tag{3.4}$$

since a is in the support of the value in (3.3).

The general form of abstraction is useful for specifying object-level binding operations whose *binding* names (rather than bound names) occur in a concrete data structure of some fixed type. For example, let us add pair expressions and patterns to the fragment of ML we have been using as a running example, by redeclaring `term` as follows:

```
# type pat =                              (* patterns, p *)
      VarPat of var                       (* x *)
    | PairPat of pat * pat;;              (* (p1,p2) *)

# type term =                             (* terms, e *)
      Var of var                          (* x *)
    | Pair of term * term                 (* (e1,e2) *)
    | Fn of <<pat>>term                   (* fn p => e *)
    | App of term * term                  (* e1 e2 *)
    | Letfun of                           (* let fun f p = e1 in e2 *)
        <<var>>((<<pat>>term) * term);;
```

For example, the ML first projection function $fn\ (x,y) \Rightarrow x$ can be represented by the value $\texttt{Fun}([\texttt{PairPat}(a,b)](\texttt{Var}\ a))$ of the above type `term`, assuming a and b are atoms of type `var`.[3]

This change to `term` hardly affects the definition of capture-avoiding substitution in Fresh O'Caml, which becomes:

```
# let rec subst e x e' =
    match e' with
        Var y -> if x = y then e else Var y
      | Pair(e1,e2) -> Pair(subst e x e1, subst e x e2)
      | Fn(<<p>>e1) -> Fn(<<p>>(subst e x e1))
      | App(e1,e2) -> App(subst e x e1, subst e x e2)
      | Letfun(<<f>>(<<p>>e1,e2)) ->
          Letfun(<<f>>(<<p>>(subst e x e1), subst e x e2));;
val subst : term -> var -> term -> term = <fun>
```

[3] Of course, linearity conditions on object-level patterns are not enforced by the declaration of `term`; so for example, $\texttt{Fun}([\texttt{PairPat}(a,a)](\texttt{Var}\ a))$ is also a value of type `term`.

3.9 Reference cells

The interactions between reference cells and the model of name binding employed by Fresh O'Caml may be surprising to the reader. For the address of a reference cell is treated as being *pure*—that is to say, it is treated as containing no atoms. It follows that any swapping operations on the address of the cell will not have any effect—in particular, *the contents of the cell are unchanged*.

We can illustrate the semantics with the following code fragment, which always produces the result `false`[4]:

```
let a = fresh;;
let x = <<a>>(ref a);;
match x with <<a'>>a'' -> a' = !a'';;
```

Compare this to the following fragment, which always produces `true`:

```
let a = fresh;;
let x = <<a>>a;;
match x with <<a'>>a'' -> a' = a'';;
```

This semantics for swapping operations on reference cells is theoretically straightforward, but perhaps pragmatically unsatisfactory. However, it is not yet clear how the situation could be improved and further research will be required.

[4] As of the time of writing (February 2005), there is a bug in the released versions of the Fresh O'Caml system which causes incorrect results when using reference cells. This will be fixed in the near future.

Bibliography

[1] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002. Special issue in honour of Rod Burstall. To appear.

[2] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml System*. Institut National de Recherche en Informatique et en Automatique (INRIA), release 3.06 edition, August 2002.

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

[4] M. R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge Computer Laboratory, 2005.

[5] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 2005. To appear; a preliminary version appears in the proceedings of the second workshop of the EU FP5 IST thematic network IST-2001-38957 APPSEM II, Tallinn, Estonia, April 2004.

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