Swapping the Atom: Programming with Binders in Fresh O'Caml

System Description for MERλIN 2003 (Category C)

Mark R. Shinwell

Cambridge University Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge, CB3 0FD, UK

Email: Mark.Shinwell@cl.cam.ac.uk Phone: +44 (0)1223 763654

Abstract

We describe Fresh O'Caml, a metalanguage equipped with facilities to manipulate object-level syntax involving α-convertible names and binding operations. The language extensions made to Objective Caml are surveyed from a practical perspective and the implementation details are briefly discussed.

1 Introduction

Fresh O'Caml aims to provide facilities for 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 standard properties of α-convertible object language names are correctly respected. Importantly, an algebraic datatype in Fresh O'Caml may involve the use of the 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.

2 Language overview

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 yielded the first version of FreshML [2], which used a complex static type system in order to guarantee correctness properties about the language. More recently, such a type system has been found to be unnecessary [4], leading to newer versions of the interpretive system available from `http://www.freshml.org/`, whose language syntax evolved from that of the Core of Standard ML. Fresh O'Caml transfers these ideas to the Objective Caml language and is available from the same web site. For a more detailed description of the

previous languages and systems we refer the reader elsewhere [4].

In the following sections we describe the extensions which have been made to O'Caml, drawing on the polymorphic lambda calculus (PLC) [3] as an example object language. Our approach differs from approaches such as the Bindlib library[1] for O'Caml which use higher-order abstract syntax internally, not least because Fresh O'Caml gives direct access to the names of bound entities.

Types of bindable names. Fresh O'Caml provides a polymorphic family `'a name` of types of *bindable names*. For each type τ, the values of type τ `name` (called *atoms*, which behave rather like values of the ML type `unit ref`) 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 name 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.[2]

For example in PLC there are two distinct classes of bindable names: type variables and variables, each α-convertible within their own class. We can declare Fresh O'Caml types `tyvar` and `var` whose values correspond to these names as follows. Note how we use the O'Caml abstract type declarations such as `type t` as a generative source of type parameters for the type constructor `name`.

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

Abstraction values and types. For expressions e, type expressions *ty* and patterns *pat*, object-level binding is represented using the abstraction-former <<e>>e' which constructs values of type <<*ty*>>*ty*$'$ known as abstraction values. Such values are *abstract* in the sense that they cannot be inspected directly; instead, one uses the abstraction pattern <<*pat*>>*pat*$'$ to deconstruct them. Fresh O'Caml ensures that each time an abstraction is deconstructed, any bound entities inside are made fresh with respect to the current environment – thus ensuring that name clashes do not occur and furthermore providing the user with access to *concrete* names throughout their terms when required.

Abstraction types may occur in datatype declarations such as the following which represent PLC types and terms, showing the power of the language for correctly representing object language syntax trees:

```
type typ  = TYvar of tyvar
          | TYfn of typ * typ
          | TYforall of <<tyvar>>typ;;

type term = Tvar of var
          | Tlam of typ * <<var>>term
          | Tapp of term * term
          | Tgen of <<tyvar>>term
          | Tspec of term * typ;;
```

For example, the PLC term $\Lambda\alpha\ .\ \lambda x\ :\ \alpha\ .\ x$ could be constructed as follows, using the self-evident translation from PLC to values of the above datatypes:

[1] `www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/bindlib.html`

[2] This is a simpler approach than the `bindable_type` declaration in [4].

```
let alpha, x = fresh, fresh in
    Tgen (<<alpha>>(Tlam (TYvar alpha, <<x>>(Tvar x))));;
```

The system prints the value as follows.

```
Tgen <<name_0>>(Tlam (TYvar name_0, <<name_1>>(Tvar name_1)))
```

Note how the numbered names show the object-level binding clearly. 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.

The important correctness property of FreshML discussed at length in [4], specialised to this example, is that *Fresh O'Caml values of type* `typ` *(resp.* `term`*) are observationally equivalent iff they correspond to* α*-equivalent PLC types (resp. terms)*. Thus we obtain working "up to α-conversion" for free. Meta-level equality of values corresponds directly to object-level α-equivalence; for example, if e_1 and e_2 are expressions of type `term`, then the boolean-valued expression e_1`=`e_2 evaluates to `true` just in case e_1 and e_2 evaluate to values representing α-equivalent PLC terms.

Fresh O'Caml permits expressions e of arbitrary type in binding position[3] in an abstraction expression, so long as the value to which e evaluates is comparable (may be tested for equality). For example, one might choose to represent the simultaneous binding of a set of object-level identifiers such as in the Scheme `let`-expression `(let ((`x_1 e_1`) ... (`x_n e_n`)) `b`)` simply by constructing an expression of the form `<<[`x_1`; ... ; `x_n`]>>([`e_1`; ... ; `e_n`], `b`)`.

Abstraction patterns. Deconstruction of an abstraction value is performed using an abstraction pattern. 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, the system ensures that any bindable names which occur inside these values and are bound by an abstraction are suitably freshened before the values are returned to the user. This is achieved by consistently *swapping* these existing atoms inside the values with as many fresh ones as necessary. This deconstruction procedure 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. This can be seen in the third clause of the following function which substitutes a type for a type variable throughout a PLC type.

```
let rec tySub ty v ty' =
    match ty' with
        TYvar v' ->
            if v = v' then ty else TYvar v'
      | TYfn (ty1, ty2) ->
            TYfn (tySub ty v ty1, tySub ty v ty2)
      | TYforall (<<a>>ty1) ->
            TYforall (<<a>>(tySub ty v ty1));;
```

Atom-swapping. The linearity constraint on O'Caml patterns which forbids multiple occurrences of the same variable means that it is not possible to write a function of the form `let f <<a>>x <<a>>y = ...` where multiple abstraction values are concreted at the same set of fresh atoms. Not only are such pattern-matches required in order to encode certain bijections which hold in the underlying FM-set theory (for example between `<<'a name>>'b * <<'a name>>'c` and `<<'a name>>('b * 'c)`),

[3] That is, within the double angle brackets.

but have also been found to be necessary during practical experimentation with the language.

As a result, Fresh O'Caml provides a swap-expression which explicitly swaps all occurrences of two particular atoms inside a given value. Having pattern-matched against a pair of abstraction values, it is possible to use such an expression to swap the new concrete names in the second deconstructed value with those occurring similarly in the first. The result is two abstraction values concreted at the same set of names: precisely what is required in order to simulate a non-linear matching. We use the above isomorphism as an example.

```
let f (<<a>>x, <<b>>y) = <<a>>(x, swap a and b in y);;
```

Fresh-for test. The boolean-valued language construct e `freshfor` e' is true iff e is a name and evaluates to an atom which occurs in the *support* of the value to which e' evaluates. 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 bindable names in that term. The fresh-for test thus provides a built-in "calculate the free variables of an object language term" function.

3 Implementation details

The O'Caml compiler patch consists of straightforward modifications to the lexer, parser, type-checker and associated utility modules. A set of low-level C primitives provides runtime support for the majority of the new features; the bytecode compiler is patched to generate code to call these primitives when required. With the exception of adding new block tags for atoms and abstraction values, no modification has been made to the actual layout of values. Atoms themselves are allocated in such a way that unreferenced ones are automatically reclaimed by the garbage collector.

More efficient implementations of Fresh O'Caml would be possible: for example the atom-swaps caused by pattern-matching against abstraction values could be delayed until the latest possible time rather than being applied immediately. This has been shown to work correctly in the FreshML interpretive system but has not yet been ported to Fresh O'Caml.

4 Conclusions and future work

Our previous experiences programming with FreshML are very promising and indicate that it provides a good paradigm for programming with binders. It is hoped that Fresh O'Caml will enable more users to experiment with the new facilities and produce real-world programs using them.

On the implementation side there are many possible avenues of research, including work on:

- a more efficient implementation using 'delayed swapping';
- compilation to native code;
- an improved pattern-matching algorithm for abstraction patterns.

On the 'user' side, it is necessary to experiment further with larger-scale software systems written in the language to determine the utility of the new features. Finally, the underlying theory will need continued development, with the aim of ensuring that any new features added to Fresh O'Caml are provably correct.

5 References

[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.

[2] 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.

[3] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Berlin, 1974.

[4] 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 ?–? ACM Press, August 2003.