

Fresh O'Caml: nominal abstract syntax for the masses

Mark Shinwell

University of Cambridge Computer Laboratory

ML Workshop 2005

Introduction

Fresh O'CamL is an extension of O'CamL designed for manipulating syntax involving bound variables.

Here in Tallinn last April I described work with Pitts that showed:

- how to give a denotational semantics to the core of Fresh O'CamL;
- and in particular how to use a *continuation monad* to model dynamic allocation of names.

This talk is more pragmatic. I will discuss:

- some design choices arising from Fresh O'CamL in use;
- algorithms used to implement nominal abstract syntax;
- the current state of the implementation;
- and what lessons have been learnt from the whole project.

Successes of existing versions

Fresh O'CamL was probably the first full-scale programming language with inbuilt support for handling binding.

Particular plus points are:

- Lightweight but powerful syntax (particularly pattern-matches on abstraction values).
- Datatypes representing syntax *up to α -equivalence*.
- The solid mathematical foundation.
- Compilation to both bytecode and native code.
- Its ease of use for rapid prototyping.

Fresh O'CamL uses *first-order* representations of binding: it does not use higher-order abstract syntax.

What we've got at the moment

- A family of types `'a name` for representing object-level names.
- The `fresh` expression for creating new *atoms*.
 - Atoms are the values which represent object-level names.
- Expressions `<<exp>>exp` for representing object-level binding operations.
 - These are called *abstraction expressions*.
 - Correspondingly, we have abstraction values and types.
- A means of pulling apart abstractions via pattern-matching.
 - This is the *only* way of looking inside an abstraction.
- Some utility functions (`swap`, `freshfor`, etc).

Very quick example

The following represents λ -terms *up to α -equivalence*:

```
type t and var = t name
type lam = Var of var
          | Lam of <<var>>lam
          | App of lam * lam
```

Here is a fragment to do *capture-avoiding* substitution $[t/x]t'$:

```
let rec subst t x t' = match t' with
  Var y -> if x = y then t else t'
  | Lam (<<bv>>body) -> Lam (<<bv>>(subst t x body))
  | App (t1, t2) -> App (subst t x t1, subst t x t2)
```

If you want capturing substitution, you can do that too...

So, what's the problem?

- Syntax becomes clumsy when describing complicated binding structures.
- Some binding structures are *impossible* to describe.
- No ordering or hashing permitted on atoms.
- Poor efficiency when pattern-matching.
- Incorrect treatment of mutable state.
- Crashes when dealing with cyclic structures.
- No support for emission of 'generic' code (capture-avoiding substitution, etc).

Time for a new release. . .

Aims of the new version

Some of the main aims of the new version are:

- To attempt to improve efficiency.
- Permit more expressive binding specifications.
- Investigate hash functions and orderings on names.
- Correctly handle cyclic data structures.
- Increase robustness.

It must be emphasised that the new version is still a work in progress so not everything is yet finalised.

Improving efficiency

- Pattern-matches on abstraction values have poor time complexity.

```
let x = fresh var in
let y = fresh var in
let t = <<x>>(…<<y>>(…(x, y)…)) in
  match t with
    <<x>>t -> …
```

- My PhD thesis proposes a scheme of *explicit permutations*, based on an idea by Mark Shields, to solve this problem.
- Implementation of explicit renamings seems impractical in the general case.
- The current plan is to delay renamings only at abstraction nodes.
 - Still means pattern matching is not $O(\text{patterns})$, but it removes some of the worst cases.
 - *Purity analysis* could help too.

Enhanced notions of abstraction – 1

- The current syntax for abstraction is somewhat barbarous.

- Non-recursive let bindings:

```
let x : int = e1
    y : int = e2 in body
```

might translate to:

```
Elet ([e1; e2], <<[(x, Tint); (y, Tint)]>>body)
```

- Recursive let bindings:

```
let rec f x = ...x...g...
    and g y = ...f...y... in body
```

might translate to:

```
Eletrec (<<[f; g]>>(<<x>>...x...g...;
                <<y>>...f...y...], body))
```

- Is this *really* a problem? Opinions differ. What certainly is a problem is the inability to express certain forms.

Enhanced notions of abstraction – 2

- More troublesome examples are those where we must only bind a *specific variety* of names.
- The new version of Fresh O’Caml provides *restricted abstractions* to help with this.
- For example:

```
type pat = Pvar of var * ty
         | Precord of (label * ty) list
```

```
type term = Evar of var
           | Efn of ty * <<var>>term
           | Etyfn of ty * <<tyvar>>term
           | ...
           | Elet of term * <<pat//var>>term
```

Restricted abstractions

- Restricted abstractions don't play as nicely as we might like. For example:

```
# let f <<a>>x = (a, x);;  
- f : <<'a>>'b -> 'a * 'b = <fun>  
# let a = fresh var;;  
- a : var name = name_0  
# f (<<a//tyvar>>a);;
```

- Currently, 'normal' abstractions and restricted abstractions are kept strictly separate for the purposes of typing, etc.
- Since we lack dynamic type information, we also forbid the construction of restricted abstractions like this one:

```
let f (x : 'a) y = <<x//'a>>y
```

Non-linear pattern matches

- Complicated forms of binding lead to complicated ways of pattern-matching. We would like to write:

```
let rec termeq t t' = match (t, t') with
...
| (Elet (t1, <<p>>t2), Elet (t1', <<p>>t2')) ->
  termeq t1 t1' && termeq t2 t2'
...
```

- The new version of Fresh O'Caml provides a new standard library function to help with this:

```
Freshness.match :
  (<<'a>>'b) -> (<<'a>>'b) -> ('a * 'b * 'a * 'b)
```

- Unfortunately, another version is needed for restricted abstractions...

Putting an ordering on the atoms

- It is important to order atoms, or provide facilities for hashing atoms, so that we can put them into maps, sets, etc.
- The operation of generating fresh names must not disturb the existing order.
- Currently Fresh O’Caml only provides ‘unsafe’ ways of hashing atoms to names.
- Why is exposing the internal order a big deal?
 - It is not completely clear that it will not break the correctness properties.
- We are starting to work on a theory which may help in this area.
 - Mostowski-style models, with dense linear orders and order-preserving permutations, seem not to be suited to our application.

Handling of textual names

At many points in a ‘real’ metaprogram we will need to deal with *textual* names rather than atoms.

- During parsing, automatic support is desirable to establish a map between textual names and atoms.
 - Such maps can be used for error reporting, pretty-printing, etc.
- However, tagging all atoms with a textual identifier is thorny: what should the semantics be on pattern-matches, for example?
 - We cannot allow direct access to the innards of abstraction values without freshening.
 - Programmers may well desire different policies for different applications.
 - Even if this were worked out, there is still the issue of having an ordering on the atoms...

Implementation

- Fresh O’Caml is engineered on top of a standard Objective Caml distribution.
- Most of the ‘Fresh’ parts are written in C.
 - Out of about 22,000 lines of runtime code, 4,000 are specific to Fresh O’Caml.
 - Modifications to the parts of the compiler written in O’Caml are much smaller, although tricky in parts.
 - Hopefully some of the C parts can be rewritten in ML.
- By and large, most of the changes made to the compiler are orthogonal to the rest of the code.

Fresh renaming

Fresh renaming is one of the key pieces of functionality provided by the runtime system. It is difficult to implement in C:

- 1 Values must be traversed using a heap-allocated stack.
- 2 We must watch out for cyclic values.
- 3 To copy parts of values, we need to allocate on the O'Caml heap.
 - This causes minor garbage collections...
 - ...and thus disturbs algorithms for detecting cycles.
- 4 In some cases we *must* break sharing, but in other cases we would like to preserve it.

How to do fresh renaming

Fresh renaming may be implemented in the presence of a garbage collector using a three-pass algorithm:

- 1 Traverse the heap graph to determine which allocations need to be performed.
 - This requires no allocation on the ML heap.
 - We can therefore use a hashtable to detect cycles and sharing.
- 2 Allocate the blocks on the ML heap and store their addresses in the C heap.
- 3 Traverse the heap graph again, filling the newly-allocated blocks and performing renamings.

Sharing: a trap

- Whilst it is desirable to preserve sharing, in some cases we *must* break it.
- In the current implementation this situation arises when freshening values during pattern-matching. For example:

```
let a = fresh var
let abst = <<a>>a
let pair = (abst, abst)
match pair with (<<x>>y, <<x'>>y') -> x = x'
```

- Preserving sharing, in general, in such situations seems hard.

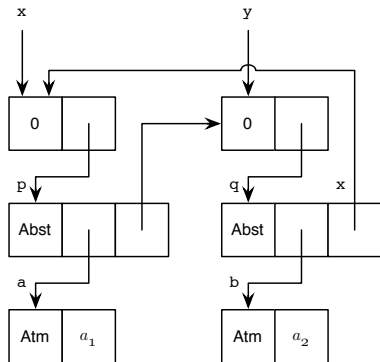
Algebraic support

- Calculating algebraic support corresponds to calculating free variables at the object level.
- O'Caml's capability for cyclic structures leads to tricky situations:

```
let a = fresh var
let b = fresh var

type t = C of <<var>>t

let rec x = C (<<a>>y)
    and y = C (<<b>>x)
```



Why algebraic support is hard to calculate

- We traverse the heap graph to derive a set of equations which can be solved to give the desired algebraic support:

$$\text{supp}(x) = \text{supp}(p)$$

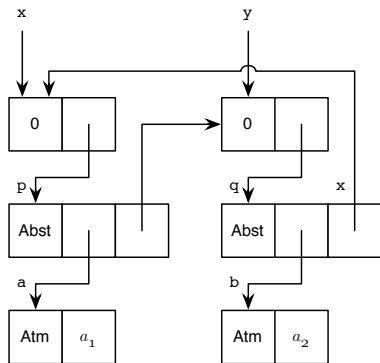
$$\text{supp}(y) = \text{supp}(q)$$

$$\text{supp}(p) = \text{supp}(y) - \text{supp}(a)$$

$$\text{supp}(q) = \text{supp}(x) - \text{supp}(b)$$

$$\text{supp}(a) = \{a_1\}$$

$$\text{supp}(b) = \{a_2\}$$



Solving the constraints

- Well-formed abstraction values lead to monotonic constraint operators.
- Here is an example of how to build an ill-formed abstraction:

```
let a = fresh var
let rec x = <<x>>a
```

- Situations such as these should be ruled out at compile-time.

Drawbacks of a patched compiler

- Issues of *fragility*: how the patch is affected by changes to the O'Caml compiler.
 - Porting to a new version of O'Caml takes about an hour.
 - Much of the patch is extra C source files.
 - The multitudinous other changes can usually be automatically applied.
- Issues of *compatibility*: are Fresh O'Caml object files compatible with O'Caml object files and C libraries?
 - They are, modulo a small restriction on the number of data constructors.
 - `bindable_type` declarations are encoded using a trick.
- Issues of *maintenance*: someone needs to keep it in sync with the O'Caml releases.

Things we don't know how to do

- Support for general delayed renamings.
 - Perhaps just delaying them at abstraction nodes, maybe in conjunction with some ad-hoc optimisations, is enough.
- A 'useful' treatment of mutable state.
 - This means not treating reference addresses just like integers. . .
- Theories of ordered names (but we're working on it).

There are also many other improvements which could be made (for example, an improved binding specification language).

The arrival of some friendly competition has stimulated debate. . .

New kid on the block



... for as you've just heard, Caml addresses the same problem area.

Comparison

- $\text{C}\alpha\text{ml}$ provides similar functionality to Fresh O'CamL, but is written as a preprocessor.
- When comparing the two methods of implementation, there are lots of issues to consider:
 - Elegance: bespoke compiler or preprocessor.
 - Syntax: abstraction, pattern-matching, etc.
 - Mathematical underpinnings.
 - Expressiveness of the binding specification language.
 - Requirements for dynamic type information.
 - Scope for improving efficiency.
 - Treatment of cyclic structures.
 - Generation of generic code.
 - Ease of maintenance.
- Perhaps the two approaches could be combined into something more powerful?

Conclusions

- Fresh O'Caml seems to have been fairly successful.
 - Many issues in nominal abstract syntax have been explored.
 - It has inspired the construction of $C_{\alpha}ml$.
- However, a robust, full-fledged implementation is proving harder than expected.
- Work continues, both practical and theoretical.
- For more information and software see the website:
`http://www.fresh-ocaml.org/`
- Next release should be out by the end of the year.