

Fresh O’Caml: nominal abstract syntax for the masses

Mark R. Shinwell¹

University of Cambridge Computer Laboratory, Cambridge, CB3 0FD, UK

Abstract

Nominal abstract syntax, as pioneered by the ‘FreshML’ series of metalanguages, provides first-order tools for the representation and manipulation of syntax involving bound names, binding operations and α -equivalence. Fresh O’Caml fuses nominal abstract syntax with the full Objective Caml language to yield a functional programming language with powerful facilities for representing and manipulating syntax. In this paper, we first provide an examples-driven overview of the language and its functionality. Then we proceed to comment on some of the difficult issues involved in implementing nominal abstract syntax and explain how they have been addressed in the latest version of the compiler.

Key words: Names, name binding, α -conversion, metaprogramming

1 Introduction

It is well known that the writing of programs to represent and manipulate syntax involving binding operations is less than trivial. This paper provides a work-in-progress report that describes the current state of Fresh O’Caml—an extension of the Objective Caml programming language [1] which provides inbuilt support for such metaprogramming tasks.

Fresh O’Caml² uses the technique of *nominal abstract syntax* pioneered by the ‘FreshML’ language designs [17,21], which in turn were based on original work by Pitts and Gabbay [12,11] in the setting of *FM-sets*. This theory in itself has given rise to a number of interesting avenues of research: for example, work on *nominal logic* [16] and logic programming [9]; denotational models involving the use of continuation monads to model dynamic allocation [20,6]; and fully-abstract models based on game semantics [3]. It has also led to the development of other language tools that address the same problems as Fresh

¹ Email: Mark.Shinwell@cl.cam.ac.uk

² Available for download from <http://www.fresh-ocaml.org/>

O’Caml, in particular Pottier’s Caml [18] and Cheney’s Haskell library known as FreshLib [8]. In common with these tools, but unlike alternative approaches such as those based on the use of *higher-order abstract syntax* (HOAS) [15,13], Fresh O’Caml does not use the function abstraction constructs of the metalanguage to represent binding operations. From the point of view of a functional programming language, this is convenient as it means that familiar principles of structural recursion may be retained.

Significant motivation and a theoretical basis for the general structure of the Fresh O’Caml language may be found elsewhere [20]. Rather than addressing this here, we seek to describe current features provided by the language—some of which are very new—and discuss how some of the problems encountered during their implementation have been solved.

Fresh O’Caml is fully incorporated into the Objective Caml system, unlike Caml which largely acts as a preprocessor and FreshLib which is wholly contained within a library module. The prime reason for Fresh O’Caml taking this approach (in many ways the most difficult option) was primarily so that the language could act as a testbed to discover the successes and failures arising when nominal abstract syntax is tightly incorporated into a language design. A second reason for producing a completely integrated system relates to the evolution of Fresh O’Caml, for it is one in a line of languages which arose directly from a mathematical model: that of ‘FM-sets and finitely-supported functions between them’. Whilst we do not need to discuss what this means here, the key idea is that it is natural that the new constructs (such as facilities for representing object-level binding) introduced in Fresh O’Caml should be just as integrated as those (such as sums and products) arising from more familiar models based on sets and functions. One example where this shows through is in Fresh O’Caml’s clean support for nested pattern matches on values representing binding constructs—something provided neither by FreshLib nor Caml.

It is important to note that whilst Fresh O’Caml most certainly constitutes a wholesale extension of the O’Caml compiler, *parts of programs compiled with Fresh O’Caml which do not utilise the ‘Fresh’ features run at the same speed and use the same space as when compiled using the standard O’Caml system.*

The language described in this paper is that which will be accepted by the next version of the Fresh O’Caml compiler, scheduled for release in the autumn of 2005. Implementation work for this is well under way, but it is still possible that language details may change before the release.

2 Language overview

In this section we use examples to show how the terms of some *object language* may be manipulated in the Fresh O’Caml metalanguage. As an example object language we adopt a version of F_{λ} ; [7] with records and pattern-matching as used in the POPLMARK challenge [4]; the reader is referred to that liter-

ature for a more comprehensive description than we have space to give here. Terms of the language, which are largely self-explanatory, are generated by the following grammar; x ranges over term variables (called *pattern variables* when they occur inside a pattern), α over type variables and l over field labels.

$$\begin{aligned} \text{types, } \tau ::= & \alpha \mid \text{Top} \mid \tau \rightarrow \tau \mid \forall \alpha <: \tau. \tau \mid \{l=\tau, \dots\} \\ \text{terms, } t ::= & x \mid \lambda x : \tau. t \mid t t \mid \lambda \alpha <: \tau. t \mid t [\tau] \mid \\ & \{l=t, \dots\} \mid t.l \mid \text{let } p=t \text{ in } t \\ \text{patterns, } p ::= & x : \tau \mid \{l=p, \dots\} \end{aligned}$$

In a universal type $\forall \alpha <: \tau. \tau'$, the type variable α is bound in τ' but not in τ . Similarly in $\lambda \alpha <: \tau. t$, α is bound in t but not in τ . In $\text{let } p=t \text{ in } t'$, all *pattern* variables inside p are bound in t' (but not t); however, any *type* variables occurring inside p are bound neither in t nor t' .

Types and terms are identified, as is usual, up to α -conversion. Since we are primarily interested in how α -equivalence classes of such syntactic entities can be represented in Fresh O'CamL, we are *not* going to identify record types, terms and patterns up to permuting the order of their fields. This saves the complications introduced by a structural congruence relation and facilitates a more lucid presentation.

2.1 Bindable names

We use the phrase *bindable name* to mean an object language name which is able to take part in binding operations. In the syntax of some object language, there may be more than one variety of such names; the names belonging to a particular variety must be kept suitably separate for the purposes of α -conversion. For example, in $F_{<}$, we have two varieties of bindable names: term variables and type variables. Each of these varieties must be declared to Fresh O'CamL using a `bindable_type` declaration³. For our example language, we simply issue:

```
bindable_type var
bindable_type tyvar
```

and thus obtain two *types of bindable names*, `var` and `tyvar`, with which to work. Note that for implementation reasons which we shall touch on later, the `bindable_type` declaration is not generative⁴. In a situation where more

³ This supercedes the rather stilted declarations of the form `type t and var = t name` which were necessary in earlier versions of Fresh O'CamL. Apart from being a somewhat clumsy hack, this means of faking a type class is tedious in systems built from multiple modules since both declarations need to be replicated in signatures. Unfortunately, the new scheme is significantly more difficult to implement.

⁴ That is to say, types of bindable names are identified by their paths rather than by unique stamps that are dynamically-generated whenever a `bindable_type` declaration is issued.

than one identical such declaration may be issued (at the toplevel, for example), issuing multiple identical `bindable_type` declarations simply causes the compiler to produce a warning and do nothing else.

Fresh O’Caml provides polymorphism over bindable names using a class of *bindable type variables* in the style of FreshML [21]. This replaces the family of types `'a name` found in previous versions of the language. A bindable type variable, written `'@a`, `'@b`, etc, may only be unified with a type of bindable names, another bindable type variable, or a non-bindable type variable (which will be forced to become a bindable one).

Having declared the various types of bindable names which we need, the creation of values of those types may be accomplished using `fresh` expressions. Values of a type of bindable names are called *atoms*, so-called because they correspond to entities of the same name [20, §4.1] in the intended denotational semantics [20, §5]. When using the `fresh` expression to create a new atom—one which will be distinct from any created previously—we must specify the (concrete) type of bindable names for it. For example:

```
let x = fresh var
```

Previous versions of Fresh O’Caml did not require such type specifications, but they are now needed since we must be able to determine any particular atom’s type of bindable names at runtime. This is needed to implement the new *restricted abstraction* constructs which we describe later.

Atoms are really quite abstract things: the only comparison operation, for example, which may safely be performed upon them is that of an equality test. Other ordering tests are exposed as ‘unsafe’ operations, since it is not possible to put the atoms in bijection with sets such as the ordered naturals inside the intended denotational semantics⁵.

The lack of a known-safe ordering is inconvenient when trying to implement efficient data structures keyed on atoms, where an ordering may be required. It is possible that the exposure of the ordering to the programmer does not break the correctness properties of Fresh O’Caml (which we shall touch on below and are explained at length elsewhere [20]); however, this is not at all clear. Currently, research is underway to determine whether the core features of the language can be given either a fully-operational semantics or alternatively a denotational one based in a slightly different setting⁶. Such a semantics—free

⁵ Such a construction is not *finitely supported* [20, §4.1] and thus does not correspond to any function in the universe of FM-sets.

⁶ For readers who are familiar with the theoretical basis of our work, we now sketch a few recent thoughts on the subject of models involving ordered atoms. In order to formalise such a system, one possible approach might be to use a *Mostowski linearly-ordered model* [14], which works with rational numbers (for the atoms) in the usual dense linear order and order-preserving permutations upon them. It appears that such permutations can express some notion of *fresh renaming* by virtue of the density of the order. This is of interest since fresh renaming suffices to express the semantics of the useful core of Fresh O’Caml, as we shall describe later in this paper. However, it is far from clear how to use such a

as it would be from the ordering restrictions which we have at present—should provide a route to determine whether it is sound to expose an ordering on the atoms; however, previous work suggests that any proof of such a result is likely to be somewhat substantial.

2.2 Representing binding operations

The type grammar of $F_{<}$ may be represented in Fresh O’Caml using the following declaration.

```
type label = string

type ty = Tvar of tyvar | Ttop | Tfn of ty * ty
        | Tall of ty * <<tyvar>>ty
        | Trecord of (label * ty) list
```

Apart from the construction involving the double angle brackets, this is standard. What the $\ll\text{tyvar}\gg\text{ty}$ part tells the compiler is that we wish to represent a (type variable, type)-pair where the type variable is bound inside the type. This is an example of an *abstraction type*, values of which correspond to elements of *abstraction FM-cppos* [20, §4.2.23] in the denotational semantics. The crucial property is that values of type `ty` are in bijection with the α -equivalence classes of $F_{<}$ types; Fresh O’Caml structural equality coincides with object-level α -equivalence. The same goes for any similar encoding of syntax using algebraic datatypes (those not utilising the function space constructor, but possibly utilising the abstraction type constructor) in Fresh O’Caml. That this is so may be proved (along with additional correctness results) by using a denotational semantics based on continuations [20, §3–§5].

Abstraction values are easy to construct. For example, to make a representation `ty1` of the type $\forall\alpha <: \text{Top}. \alpha \rightarrow \alpha$ we simply issue the following.

```
let a = fresh tyvar
let ty1 = Tall (Ttop, <<a>>(Tfn (Tvar a, Tvar a)))
```

The *abstraction expression*, again written using double angle brackets⁷, is a constant-time operation when the part in *binding position*—between the angle brackets—is of a type of bindable names. Expressions of more complicated

model to construct sets or cpos to correspond to sets of values where one atom is bound (aka. abstraction FM-cppos). Instead, it may well be more appropriate to first consider an (operational) ‘possible worlds’ model, working on integers with the usual order and injective functions which preserve that order. That may provide insights into what the correct denotational model might be: if such work were to yield one based on functor categories then it would be enlightening to consider whether there is an Mostowski-style analogue to it (in the same way that FM-sets correspond to pullback-preserving functors from an index category to sets).

⁷ It should be noted that whilst the interoperation of Camlp4 [10] and Fresh O’Caml has not yet been subject to scrutiny, the alternative form `<| – |> –` may be used in place of double angle brackets to avoid a syntactic clash with its quoting mechanism.

types are permitted in binding position, as we shall see, but even then the construction is relatively fast (and still constant-time in the size of the value in *body position*—the one after the closing double angle brackets).

As a convenience, Fresh O’Caml now provides syntactic sugar to further reduce clutter when constructing complicated object language terms. This enables us to rewrite *both* lines above into a single one as follows:

```
let ty1 = Tall (Ttop, ‘‘a. Tfn (Tvar a, Tvar a))
```

Note that the double backquotes act as a meta-level binding operation, unlike the abstraction expression which is *not* a binder.

So what happens once an abstraction value has been created? Such values act like black boxes: they package up a value in binding position together with the corresponding value for the body, and may only be deconstructed in a manner so as to preserve the so-called *Barendregt variable convention* [5]. In particular, the *particular value* currently being used to represent the part in binding position is completely hidden by the system⁸ so as to maintain the ‘anonymity’ afforded by an α -equivalence class. Each time an abstraction value is deconstructed, the binding and body parts are suitably *freshened* so that no name clashes may occur.

Deconstruction of an abstraction value may only be performed by using a pattern match. We are going to illustrate how this works by using the following function, which performs *capture-avoiding* substitution of a type for a type variable throughout an F_{\leq} type.

```
let rec substitute_type ty' tv ty =
  match ty with
  | Tvar tv' -> if tv = tv' then ty' else ty
  | Ttop -> Ttop
  | Tfn (ty1, ty2) ->
    Tfn (substitute_type ty' tv ty1,
        substitute_type ty' tv ty2)
  | Tall (ty1, <<tv'>>ty2) ->
    Tall (substitute_type ty' tv ty1,
         <<tv'>>(substitute_type ty' tv ty2))
  | Trecord fields ->
    Trecord (List.map (substitute_type ty' tv) fields)
```

In the clause for Tall we see an example of an *abstraction pattern*: the final piece of syntax which uses the double angle brackets. If this clause is taken, the runtime system takes the scrutinee (in this case **ty**) and splits the abstraction value into the binding part (call it the atom *a*) and the body part. Then, a fresh atom *b* is chosen and mapped to **tv'**. The body pattern variable **ty2** is then mapped to the value formed by renaming all occurrences of *a* to *b* throughout the body part. In this way, the programmer only ever sees

⁸ Barring use of unsafe features such as the Obj module, of course.

freshened versions of the abstractions and no name clashes occur.

The facilities provided for pattern-matching against abstractions enable us to penetrate deep into the structure of some particular value in one fell swoop: it is perfectly acceptable to write pattern matches which contain nested abstractions. Alternative approaches such as Caml and FreshLib do not provide such functionality.

2.3 Fine-grained control of binding

We now consider how to represent pieces of F_{λ} syntax involving the `let` construct (and the associated record creation and projection expressions) in Fresh O’Caml. This piece of syntax possesses a non-trivial binding structure: recall that in `let $p=t$ in t'` only pattern variables inside p are bound in t' . What we would like to do is to express this in the metalanguage as cleanly and concisely as possible. A good start is to declare a type `pat`, for patterns and another, `term`, for terms, in the following manner.

```

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
          | Eapp of term * term
          | Etyapp of term * ty
          | Erecord of (label * term) list
          | Eproj of term * label
          | Elet of term * <<pat||var>>term

```

Note how abstraction types have been used to correspond to the object-level binding present in the two varieties of lambda-abstraction. What is more interesting is the type `<<pat||var>>term`, which is an example of a *restricted abstraction type*. Such a type in general takes the form `<< τ_1 || τ_2 >> τ_3` , where τ_1 is a type whose values are comparable, τ_2 is a type of bindable names, and τ_3 is any type. These types are new to the current version of Fresh O’Caml and permit us to tell the compiler that we wish to represent an operation where the binding names are those free object-level names of type τ_2 in the term represented by the value of type τ_1 . In the concrete example above, where we see `<<pat||var>>term`, we are telling the compiler that we wish to place a pattern in binding position yet only use its constituent `vars` as binding occurrences. This means that any type variables in such a pattern will not be treated up to α -conversion, as we desire.

In fact, Fresh O’Caml permits any comparable type to be placed in binding position within an abstraction type whether or not it is restricted by specifying the additional type of bindable names.

Deconstruction of a value formed using a restricted abstraction (whose expression syntax is identical to the type syntax, as for normal abstractions) is performed via a pattern-match in the way which might be expected, for example:

```
let f t = match t with
  ...
  | Elet (t, <<p>>t') -> ...
```

Note that pattern-matches do not specify the type of bindable names at which the abstraction is restricted. When a restricted abstraction value is deconstructed—for example a value of type `<<pat||var>>exp`—the runtime system will only cause values of type `var` to be freshened throughout the abstraction value (that information is stored in the value itself).

Now the above encoding for $F_{<}$: patterns and terms works, but as has been observed elsewhere [20,18] in similar scenarios, it is somewhat clumsy. What we would really like is to have the first component of an `Elet` data value to come after the pattern component, just as it does in the syntax. We cannot just do this blindly since it would be erroneous: then, any atoms of type `var` inside that first component would be ‘captured’ by the abstraction value and (incorrectly) treated up to α -equivalence.

One might argue that in this case we are just picking nits, but the situation gets significantly worse in the case where the object language permits more complicated binding structures. For example, suppose we wish to treat constructs like

```
let x1 = f y
    x2 = g z
in
  h x1 x2
```

where the bindings are supposed to be non-recursive. At the moment, we would have to use a data type containing a constructor declaration of the following form.

```
...
| Elet of term list * <<var list>>term list
```

In this example, the first component of such a data value would hold the function calls to `f` and `g`; the binding position of the abstraction would hold the list `[x1;x2]` and the body of the abstraction would hold the call to `h`. This is not only visually unsatisfactory but also pragmatically so: it is easy to create ‘junk’, as it has been termed, by constructing an `Elet` where the lists differ in length.

The author’s PhD thesis [20, §7.1.5] raised the idea of a ‘`Nobind`’ data constructor (with associated type constructor written in lower case), the operation of which would be to enable parts of values in binding position to be treated as if they were outside the particular (innermost) abstraction enclos-

ing them. For example, we could rewrite the above fragment in the following manner.

```
...
| Elet of <<(var * (term nobind)) list>>term list
```

This at first appears to be a plausible solution to the problem; however, it is somewhat inflexible. Consider for example that we want to represent a piece of syntax such as that above but where the `let` is to be a *recursive* binding construct. Now we are stuck and `nobind` is of no use: we still have to use something like the following.

```
...
| Eletrec of <<var list>>(term list * term list)
```

Pottier’s work on Caml [18] exploits a *binding specification language* which enables both of these constructs to be expressed more effectively: we refer the reader to that work for a comprehensive explanation. In brief, it is achieved by replacing ‘bipartite’ abstraction values, consisting of a binding part and a body part, with single values whose components possess *scope specifiers*. This scheme enables a particular component of a value marked as an abstraction to either hold binding occurrences of names, to hold bound occurrences of names (using the ‘inner’ specifier) or to be excluded from binding (via the ‘outer’ specifier). The behaviour of the latter scope specifier is somewhat like `nobind`.

Such a specification language is interesting not least because it suggests ways in which Fresh O’Caml’s abstraction constructs could be improved. We discuss this further in our later section on future work.

In some circumstances, use of restricted abstraction types can be completely replaced by use of `nobind` (or, indeed, the `inner` and `outer` scope specifiers if they were to be implemented). For example, consider the following means of writing the pair of type declarations for `pat` and `term`.

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

type term = ...
          | Elet of <<(pat * (term nobind)) list>>term
          | ...
```

In this scenario, use of `nobind` to deal with patterns is not so satisfactory: it ‘infects’ the type declaration for `pat` even though a pattern, *per se*, has really nothing to do with binding operations. Restricted abstractions, on the other hand, help the properties of a particular object-level binding operation (in this case, that it only binds a particular variety of names) to be expressed at the points in the metalanguage terms corresponding to those binders. Consequentially, the version using restricted abstraction types appears to us more declarative.

As we shall identify later, the introduction of restricted abstraction types into the language has necessitated a reasonable amount of implementation work: in particular, atoms must be equipped with type representations at runtime. (The reader may wonder why this is not necessary in the first place, given that we can take a term t containing both `vars` and `tyvars` and form abstractions $\llbracket \text{var} \rrbracket t$ and $\llbracket \text{tyvar} \rrbracket t$. Such constructions do not in fact necessitate extra type information: the type system guarantees that the atom in binding position within either such an abstraction is ‘of the correct type’, and since two atoms of distinct types are always guaranteed to be structurally distinct, the bound occurrences can easily be identified throughout the term t .) The question must therefore be asked, is it worth the extra work to introduce restricted abstractions? We believe that it is. Apart from the advantages described above, it is in fact the case that it provides extra expressivity. Consider for example a syntax tree containing two varieties of names. We might wish to use this syntax tree in binding position (as in the example mentioned earlier) but yet pick only one of the two varieties of name to act as binding occurrences in one place, and the other variety in another. Restricted abstractions allow us to re-use the same syntax tree in both scenarios, which `nobind` and related constructs do not permit.

2.4 Pattern-matching: under the hood

Now that we have seen the various varieties of abstraction pattern-match—restricted or non-restricted, with just a name in binding position or with something more complicated—it is instructive to examine the matching process in more detail. To do this, we must introduce the notion of the *algebraic support* of a value, which captures the idea of ‘the atoms involved in the value’s construction’. Formally, it is a finite set of atoms approximating the *least finite support* of the denotation of the value—a notion defined elsewhere [20, §4–§5]. Intuitively, the algebraic support of some value will correspond to the *free variables* of the object language term which it encodes. Unlike the formal definition of least finite support, which may be thought of in the same way, it is calculated by a simple structural recursion⁹. (Least finite support, strictly speaking, is defined in terms of permutations of atoms.) For example, the algebraic support of an atom a is just the singleton set $\{a\}$ whilst the algebraic support of a pair (v, v') is the union of the algebraic supports of v and v' .

Given an abstraction value $\llbracket v \rrbracket v'$ then the algebraic support is calculated by taking the algebraic support of v' and subtracting the algebraic support of v . For a restricted abstraction value $\llbracket v \mid \tau \rrbracket v'$, the process is similar, except that the algebraic support is calculated by taking the union of the algebraic supports of v and v' , then removing those atoms in the algebraic support of v which have type τ .

⁹ Except where cyclic values are present, as they are in O’Caml: we explain how to cope with this later.

The case for function values is altogether more thorny. Unfortunately, the calculation of such a value’s algebraic support can only be an approximation of its (semantic) least finite support: which atoms are in the exact least finite support of the denotation of such a value is recursively undecidable. By taking the union of the algebraic supports of the free variables of a function value, we can produce a set of atoms which definitely contains the exact algebraic support, but may be an over-approximation. A simple example is given by considering the following function which has a single free variable `a` (intended to be of a type of bindable names):

```
let f x = if a <> a then x else x
```

The function has empty support, but the approximation will calculate the set $\{a\}$, where a is the atom assigned to the identifier `a`.

Given this problem, we must forbid function values in binding position, since the contravariance present in the abstraction case means that the calculated algebraic support for a binding-position value must be exact. Indeed, the current implementation of Fresh O’Caml simply raises an exception if asked to calculate the support of a function value (whether in binding position or not); it is thought that this is the solution which is likely to cause the least confusion among users. Given that such users are highly likely to be working with algebraic data types it is unlikely to be much of a restriction, if any.

Returning to the subject of pattern-matching, suppose we are matching a scrutinee $\ll v \gg v'$ against an abstraction pattern $\ll p \gg q$. The first step is to calculate the algebraic support of v . All of the atoms that land in the support set (call it ω) are known to be representing binding occurrences of names in the object language syntax. Next, we allocate as many fresh atoms ϕ (and ensure they are piecewise tagged with the same types as the ones in the support set) and fix a bijection $\psi : \omega \leftrightarrow \phi$ which respects this tagging. We then calculate two new values, p and q , to correspond to the pattern variables `p` and `q`. The value p is formed by using ψ to ‘fresh-rename’ the value v ; q is formed in the same way from v' . Note that whilst fresh renaming is an instance of the more general notion of *swapping*, upon which the denotational semantics of the core of Fresh O’Caml is based, it is all that is required to implement this scheme.

To handle pattern-matching in the case where restricted abstractions are present, we follow the same process, except that we restrict the calculated algebraic support of the value in binding position to those atoms of the correct type.

This is an appropriate point in the discussion to mention the efficiency of Fresh O’Caml. This still an ongoing concern and it is fair to say that it has not received as much attention as it perhaps ought to have. One significant concern relevant to this section is that the current method of pattern matching can exhibit quadratic time complexity (for example in a recursive function which works through a term containing nested abstractions). Theoretically, in order to restore the usual ML-style situation where the time complexity of

pattern-matching is independent of the size of the value being matched, we could adopt a scheme of *delayed permutations* such as that previously described by the author [20, §7.1.1]. This works in the style of explicit substitutions [2] and would be much more satisfactory: the act of pattern-matching against an abstraction node is reduced to a constant time operation. Unfortunately, our previous efforts to incorporate such a scheme into Fresh O’Caml have been unsuccessful. Reasons for this failure centre around implementation difficulties: for example, the scheme means that the macros used to access heap blocks might have side-effects including allocation (in the case where a permutation is to be pushed down a level). Notwithstanding further efficiency concerns, this causes trouble due to the way in which the macros are used throughout the runtime system: they are often not protected by the guards required to be safe across garbage collections (which are invoked by allocations). On another note, it is also likely that the introduction of a scheme of delayed permutations would break binary compatibility, which is highly undesirable.

One alternative approach, which would improve the running time in some situations, would be to only delay atom-swaps at abstraction values. This would likely be far easier to implement and may become a focus of attention in the future.

2.5 Equality testing, non-linear matches and swapping

We noted earlier that Fresh O’Caml’s built-in structural equality check serves as a test for object-level α -equivalence when applied to values of types such as `pat` and `term`. Whilst this is very useful, there are situations where it does not suffice, for example if the values contain finite sets or maps implemented using the standard library. For such data structures we usually have to use their own equality-testing functions rather than the generic structural equality test in order to achieve a correct result.

It is therefore useful to show how the inbuilt Fresh O’Caml equality test for a datatype such as `term` may itself be encoded in Fresh O’Caml. We leave the majority of it to the reader’s imagination and focus on one of the interesting parts: the clause for two `let`-binders. What we really want to write is something like the following.

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

Unfortunately, this piece of code is not legal Fresh O’Caml since it involves non-linear patterns (ones in which a pattern variable is repeated) in order to ensure that two abstraction values have equal values in binding position when deconstructed *at the same set of fresh atoms*. In previous releases of the compiler, we would have been forced to use explicit atom-swapping operations

and then an equality test in order to produce this effect ourselves. This is somewhat unintuitive and—in a case such as this where patterns can contain multiple variables—quite tricky to implement.

The latest version of Fresh O’Caml provides improved support for such non-linear deconstructions by means of the following standard library function.

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

This takes a pair of abstraction values and deconstructs them at the same set of fresh atoms. The return value is a 4-tuple containing the freshened value in binding position from the first abstraction, the corresponding freshened body, and the same for the second abstraction. (Note therefore that if the values in binding position are just single atoms, then the first and third components of the tuple will be structurally equal.) Using this, the fragment above can now be rewritten as follows.

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

A future line of investigation will be to determine whether the linearity restriction could be relaxed in cases such as these. This would obviate the need for `Freshness.match`.

Readers familiar with previous work on Fresh O’Caml or the underlying theory may be intrigued that we have not yet made much mention of the process of *swapping* atoms throughout values. In previous versions of the Fresh O’Caml system, there was indeed a language keyword (`swap`, unsurprisingly) which performed this operation, and one indeed might imagine that such an operation would be exposed to the programmer given its significance when giving both an operational and denotational semantics [20, §3 and §5] to Fresh O’Caml.

The fact of the matter is that practical experimentation with the language seems to point strongly in the direction that such a construct ought to be relegated to the point of being *available* for use if required, but not presented as a ‘frontline’ operation. The majority of uses seem to have been to simulate non-linear pattern matching, something which is now provided explicitly (and in a far more convenient manner) as we have seen. Use has however been made of the `swap` construct in order to improve efficiency in certain situations (during implementation of an interpreter for the Acute language [19] in particular), and hence the operation is still provided in the standard library (as `Freshness.swap` and `Freshness.swap_multiple`). Its use, however, is discouraged: in the event of future versions of Fresh O’Caml becoming significantly more efficient it might be removed.

All this having been said, the swapping of atoms is one of the key operations performed by the runtime system. Even though it now appears to us that swapping is not as fundamental to an implementation as first thought, the current runtime still revolves around it for historical reasons. Its implementation poses significant difficulties, some of which we shall detail in due course. Fresh renaming is slightly easier, but only due to a deep technical subtlety which we shall not delve into here.

3 Implementation

Systems using nominal abstract syntax have so far been implemented in three ways:

- as an integrated solution involving extensions of the compiler, standard library and runtime system (*viz.* Fresh O’Caml);
- as a preprocessor combined with support libraries (*viz.* C’aml);
- entirely as a library module (*viz.* FreshLib).

The integrated approach taken by Fresh O’Caml has proved useful in the sense that it has identified many of the limitations and difficulties which may arise when nominal abstract syntax is shoehorned into an existing language system. These will vary depending on the particular language system in question. For O’Caml some of them are as follows.

- The semantics of the name-manipulating core of Fresh O’Caml places heavy reliance on two key runtime operations: that for fresh renaming (in general, swapping) and that for calculating the algebraic supports of values. These operations, when applied on arbitrary heap blocks, are non-trivial to implement. The traversal of values, for example, must always be performed using a heap-allocated queue in order to avoid any possibility of stack overflow.
- The O’Caml language permits somewhat arbitrary cyclic values to be constructed on the heap (via the use of extended `let rec` expressions, for example). This complicates the algorithm which swaps atoms throughout a value on the heap. The added constraints imposed by the garbage collector make the implementation of this algorithm even more difficult. It would be possible to restrict the language so that users cannot construct these arbitrary cyclic structures (by removing the addition to O’Caml made some time ago that permits various forms of extended `let rec` expressions), but we do not do this since we are aiming to accept all legal O’Caml programs. (We do indeed succeed in this aim, modulo a tiny restriction which we identify in §3.2.) It should also be noted that the extended recursion provided by O’Caml is potentially useful in conjunction with nominal abstract syntax: for example it can be used to create representations of recursive function closures without the need for references.
- Values on the O’Caml heap contain almost no type information and have

very limited potential for tagging with extra information. This means that the representation of atoms—whose types may need to be determined at runtime—is not entirely straightforward.

These problems have now been solved, after quite some effort, and the next release of the Fresh O’Caml system will offer significantly more robust implementations of the key algorithms. There remain difficulties: in particular, a more efficient implementation of the pattern-matching algorithm would be desirable, but the changes required to an already complex piece of code make this a serious undertaking.

One key advantage of the Caml system and FreshLib over Fresh O’Caml is that they are able to assist with the generation of *boilerplate* code. Some specific examples which rear their ugly heads time and time again during metaprogramming tasks are the following:

- functions to calculate free variables of terms, and so forth;
- functions which take parser output, say from `ocamlyacc`, and translate textual identifier names to internal representations (say atoms);
- functions to perform capture-avoiding substitutions.

Fresh O’Caml does provides assistance with the first of these, for it has always been possible to use the `freshfor` keyword to determine whether an atom is in the algebraic support of a particular value or not; this corresponds to an object-level test for free names. The current release moves this functionality into the standard library (as `Freshness.fresh_for`) and augments this with another built-in expression, `support`, which when given a type of bindable names (fixed at compile-time) and a value returns a list of the atoms in the algebraic support of that value.

The second task which we identify above is tied up with the general issues of error-reporting and pretty-printing which manifest themselves when writing more substantial metaprograms. It is not good enough simply to take abstract syntax trees from a parser and translate the textual names therein into atoms (using some variety of finite map, for example) since access is required to the textual names at a later stage. As identified elsewhere [20, §2.8] it seems that the best solution for pretty-printing is to tag the nodes representing *binders* with the original names used at those points. They can then be re-used upon pretty-printing and renamed with primes if necessary. Similarly, during error-reporting, a particular occurrence of an atom can be matched up to its textual name using a finite map maintained during traversal of the syntax tree.

Such schemes can significantly increase the complexity of the metaprogram, but alternative approaches seem unclear. Tagging each atom with a textual identifier seems doomed to failure¹⁰: for example, what should the correct behaviour be when such an atom is packaged up into an abstraction value and

¹⁰ Except for names which do not take part in object-level binding operations, which can of course be represented only by strings without further ado.

then deconstructed twice? The system should probably take the approach that the textual identifier should be freshened each time along with the atom in order to avoid clashes: however, if the deconstructions in question arise from two sequential calls to a pretty-printing function, say, then this is probably not what the programmer would want. In other circumstances, the programmer might desire that the names be freshened automatically. It is not clear to us if any ‘correct’ behaviour exists, so for the moment we leave all of this to the programmer. It is not beyond the bounds of possibility that atoms could be identified in the future *just* by textual strings: after all, if results can be proven showing the soundness of systems involving ordered atoms, there should be no fundamental problem. Gratuitous renamings would still be likely to occur, however.

Even with the compiler system taking this rather hands-off approach to dealing with textual names at runtime, it would be desirable to have improved support for generating some of the boilerplate code which arises as a result of having to keep various maps between names and atoms. One obvious line of improvement might be to enhance parsing tools so that they are aware of nominal abstract syntax: this would remove the necessity for the programmer to maintain either global or monadically-threaded state (to map textual names written by the user into pairs of atoms and textual names, say) when writing parser descriptions that target Fresh O’Caml syntax trees. Similarly, the implementation of tasks such as the final one in the list above (capture-avoiding substitution functions) could be made less tedious by automatic assistance. The Caml system does provide support for such tasks and it seems unlikely that Fresh O’Caml could do so without adopting the same tactic of using a preprocessor.

3.1 *What is where*

In Fresh O’Caml, the name-manipulating functionality is either hard-wired into the compiler or exposed as standard library functions. The hard-wired facilities are as follows:

- the `bindable_type` declaration, for introducing new types of bindable names;
- the `<< - >>-` syntax for abstraction expressions, values and types;
- the convenience expression ‘`x. -;`’;
- the `fresh` expression, for creating new atoms at runtime;
- the `support` expression, for calculating the free variables of object language terms.

These facilities must be implemented in this way rather than being placed wholly in the standard library due to one of two reasons: either because they necessitate changes within the compiler which cannot be implemented elsewhere (as with `bindable_type` and the abstraction constructs) or because they take types as arguments (as with `fresh` and `support`). Conversely, the

following features are exposed through the standard library:

- swapping of atoms throughout values (`Freshness.swap` etc);
- the ‘fresh-for’ test, which determines whether an atom is in the algebraic support of a value (`Freshness.fresh_for`);
- deconstruction of pairs of abstraction values in such a way as to simulate a non-linear match (`Freshness.match`).

The various primitives implemented in C inside the Fresh O’Caml runtime system centre around the two key operations needed to implement nominal abstract syntax in our setting: that which calculates the algebraic supports of values on the heap, and that which performs fresh renaming (or swapping). We look at these in more detail in due course.

3.2 The drawbacks of a patched compiler

Any compiler system modelled along the same lines as Fresh O’Caml—where an actively-maintained program is patched to introduce extra functionality—faces certain drawbacks. In particular, there are issues of *fragility* (how the patch is affected by changes to the host compiler) and *compatibility* (whether object modules compiled with the host compiler and the patched system are interoperable).

The majority of the Fresh O’Caml patch¹¹ consists of extra C source files; our past experience shows that these are largely (and usually wholly) unaffected by changes introduced by the O’Caml maintainers between releases. The remainder of our patch introduces changes to source files written in O’Caml which form part of the usual release. Despite the fact that the patch touches a fair number of these files (everything from lexing through to code generation), the majority of the changes are of such a form that they can be automatically applied to a new version of O’Caml. Typically, porting the Fresh patch across an O’Caml version upgrade takes only an hour or so. We even conjecture that combining the Fresh patch with other patches to the O’Caml compiler, *per se*, would not be difficult: the most pressing issue in that area would simply be whether the combined patches produce a language that is sound!

As far as compatibility goes, object files from Fresh O’Caml and O’Caml are binary compatible in the vast majority of cases. The only troublesome cases would stem from the fact that the maximum number of data constructors permitted in any particular `type` declaration is slightly lower in Fresh O’Caml than O’Caml (since the former needs more distinguished heap block tags). Code which exceeds the Fresh O’Caml limit would have to be modified and re-compiled to ensure successful operation. We believe that such cases are likely to be rare.

¹¹ For the existing releases of the compiler this stands at a little under 5,000 lines when presented as output from `diff`.

Similarly, Fresh O’Caml code can interoperate with compiled C code designed only for a standard O’Caml runtime, so long as it respects the constructor limit above.

If the Fresh O’Caml implementation of nominal abstract syntax was deemed to be sufficiently useful, one could ask whether it should be incorporated into the standard O’Caml distribution. It would be quite improper of us to propose an answer to such a question here, since the development of the main distribution rests entirely with others, but it is clear that the level of maintenance of the Fresh code would decrease if it were to be incorporated in the main distribution (since the effort of patching would be removed).

3.3 Calculation of algebraic support

The calculation of the algebraic support of a value looks simple enough at first: one simply traverses the heap graph applying the various structural rules for algebraic support which we gave earlier. This must of course be done in an iterative fashion so as to rule out stack overflow. In general, however, the heap structure of a Fresh O’Caml value is a directed, possibly-cyclic graph. Nodes may have more than one incoming edge, since values may be shared; cycles are formed by the use of `let rec`. The possibility of sharing *and* cycles being present in the structure of a particular value introduces more difficulties than are obvious at first. For one thing, the contravariance present when calculating the algebraic support of an abstraction value means that certain values must be ruled out as being ill-formed. We might attempt to construct such a value as follows.

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

The algebraic support of `x`, which we write $\text{supp}(x)$, should be equal to that of `a`, namely the singleton set $\{a\}$, minus the algebraic support of `x`: it is clear that no finite set satisfies this equation. It is hoped that such constructions can be ruled out at compile-time by extending O’Caml’s checks on well-formed recursive definitions.

When traversing the heap structure of an arbitrary value, a hash table may be used to determine if a particular heap block has been visited before. However, this is not sufficient to detect whether the block is shared or is part of some larger cycle. In order to calculate the algebraic support of the whole value, it is therefore necessary to adopt a more sophisticated solution. We proceed by first examining the value and building sets of constraints. For example, the declarations

```
let a, b = fresh var, fresh var
type t = C of <<var>>t
let rec x = C (<<a>>y) and y = C (<<b>>x)
```

yields a heap graph from which we derive the following, where a_1 and a_2 are the atom identifiers assigned to \mathbf{a} and \mathbf{b} respectively. (The extra level of indirection via the blocks named \mathbf{p} and \mathbf{q} , which do not occur in the source text, arises solely because the values \mathbf{x} and \mathbf{y} are constructed values.)

$$\begin{aligned} \text{supp}(\mathbf{x}) &= \text{supp}(\mathbf{p}) & \text{supp}(\mathbf{y}) &= \text{supp}(\mathbf{q}) \\ \text{supp}(\mathbf{p}) &= \text{supp}(\mathbf{y}) - \text{supp}(\mathbf{a}) & \text{supp}(\mathbf{q}) &= \text{supp}(\mathbf{x}) - \text{supp}(\mathbf{b}) \\ \text{supp}(\mathbf{a}) &= \{a_1\} & \text{supp}(\mathbf{b}) &= \{a_2\}. \end{aligned}$$

To solve such constraints one could adopt an approach where equations are substituted into each other until the answer is found. A more elegant approach, however, is to observe that any constraint set arising from a well-formed value (that is to say, one whose algebraic support may be calculated) can simply be transformed into a monotonic operator on vectors of finite sets (each component of such a vector corresponds to the algebraic support of a particular heap block). This operator may be iterated, starting from a vector of empty sets, until its least fixed point is reached. Having to perform such an operation at runtime seems somewhat unusual, but we believe that in the majority of cases it does not give a drastic decrease in performance.

3.4 *Swapping and fresh renaming*

The second key runtime operation which we examine is that of swapping atoms throughout values on the heap. In Fresh O’Caml, this is used to implement fresh renaming and also the explicit swapping operations found in the **Freshness** module.

As with other operations on the graphs of heap values, traversal must be effected using a queue allocated on the C heap rather than relying on the system call stack. Unfortunately, the algorithm is further complicated by the fact that allocations on the ML heap must be performed during the process. This means that traditional means of detecting cyclic structures (for example by keeping a hashtable of visited blocks) may not be applied, since such structures are not stable under garbage collection (and nor may they be made so at reasonable cost).

These two problems conspire to make the implementation of swapping, or indeed just fresh renaming, far from trivial: an overview such as this cannot do justice to the complexities involved. In outline though, we have adopted a three-pass solution. In the first stage, the heap graph of the value concerned is traversed to discover which blocks need to be allocated. Because this requires no allocation on the ML heap, a hashtable can be safely used to detect cycles and/or sharing. In the second phase, the blocks are then allocated and their addresses stored into arrays on the C heap; in the third phase, the value’s graph is traversed again and the data therein—having had any applicable atom-swaps applied to it—is copied appropriately into the newly-allocated

blocks.

An additional flag to this algorithm enables it to freshen every abstraction throughout the value passed to it, rather than acting as a generic atom-swapper; this is used during the process of pattern-matching and also in the toplevel system when printing values for the interactive loop. When behaving in this way, the first phase of the swapping algorithm must calculate the algebraic supports of any values found to be in binding position (in order that it can know which atoms are to be replaced with fresh ones). Since the first phase must not allocate on the ML heap, this necessitates that the algorithm which calculates algebraic supports must also not allocate on the ML heap.

3.5 Bindable names

Since the representation of atoms—the semantic objects corresponding to value identifiers of bindable types—has changed dramatically since previous releases of Fresh O’Caml, we say a few words about it here. In previous versions, atoms were stored in distinguished blocks (with tag `Atom_tag`) containing a 31-bit atom identifier. The atom identifiers were generated using a pseudorandom number generator.

This 31-bit scheme is somewhat unsatisfactory: it is desirable to have ‘globally-unique’ identifiers (160-bit hashes, say) in order that atoms may be safely marshalled between different instances of the runtime. Furthermore, due to the introduction of restricted abstraction types, atoms must be tagged with a type representation that shows other parts of the runtime which type of bindable names they correspond to. This can be done by assigning (160-bit) *type hashes* at compile time to each type of bindable names; this hash is then stored within the atom together with its identifier. By ensuring that the `bindable_type` declaration is not generative, the process of allocating hashes to types of bindable names is made more straightforward (for care must be taken to ensure that hashes work correctly across module boundaries).

The previous 31-bit scheme could therefore be modified to allocate atoms as pairs of 160-bit words. However, this is unsatisfactory due to the necessity to reference (atom identifier, type hash)-pairs across garbage collections during operations of fresh renaming, for example. To prevent unnecessary overheads of registering global value pointers with the garbage collector, we instead allocate such pairs on the C heap. Then, atoms are represented using blocks with tag `Custom_tag` and a distinguished string identifier; the data inside the block consists of a single pointer to the corresponding block on the C heap. In this way, pointers to the C heap blocks can be held across calls to the garbage collector.

Apart from simplifying code in the runtime system and improving efficiency, this approach also has other desirable properties. For example, code to marshal and unmarshal atoms can be confined to the new runtime modules, whereas previously it would have been embedded inside existing runtime

modules. We thus achieve greater code separation and save a block tag in the process (since `Atom_tag` is no longer needed).

4 Conclusions and future work

Nominal abstract syntax is spreading: in the functional programming world, there are now three significant language systems making use of it. In this paper we have described the current evolutionary state of one of them, Fresh O’Caml, and outlined some of the tricky implementation details. Work continues on coding up what we have described here: at the time of writing, the majority of the difficult parts have been completed. Much of the runtime system has been completely rewritten for the new release.

We hope that in the future it will be possible to devote more time to improving efficiency, now that the implementation of many of the core features is well-understood. So far, Fresh O’Caml has proved very valuable as a rapid prototyping tool for new languages or small interpreters: the experience on the Acute project seems to indicate that for it to be useful on a large scale then the efficiency of abstraction pattern matching must be seriously addressed. On that project, much use was made of the `fresh` and `swap` constructs in order to handle representations of binding without using abstraction values: this was partially for efficiency reasons and partially because the compiler at that time lacked support for restricted abstractions. At least the latter of those obstacles is now in the process of being removed.

It would be pleasing if the arrival of the Caml language were to provide some more insights into the efficiency aspects of nominal abstract syntax, for that system is much less constrained by the behaviour and structure of an existing runtime. It seems likely that this will be the case. Even if such insights were not to affect Fresh O’Caml, they may well prove useful by other language implementors looking to incorporate nominal abstract syntax techniques into their work.

As hinted earlier, we hope to experiment in the near future with the integration of Caml-style `inner` and `outer` scope specifiers, or even something more general which can apply across multiple levels of nested binders, into Fresh O’Caml’s abstraction types. This would enable lucid presentations of binding structures in the manner of the following datatype, which treats both recursive and non-recursive `let` bindings. Note that a scope specifier is not required in the `Efn` case, since a value of type `ty` can never contain any atoms of type `var`.

```

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

type term = Evar of var
          | Efn of <<var * ty>>term
          | Etyfn of <<tyvar * (ty outer)>>term

```

```

| Eapp of term * term
| Etyapp of term * ty
| Erecord of (label * term) list
| Eproj of term * label
| Elet of <<(pat * (term outer)) list||var>>term
| Eletrec of <<(pat * (term inner)) list||var>>term

```

Care must be taken when considering the semantics of abstraction values which may themselves have abstractions nested in their binding-position part. At the present time we believe that it is sound to adopt a semantics where scope specifiers only have meaning in the binding position of an abstraction value and apply only to the innermost abstraction (if any) enclosing them—whether or not they occur in binding or body position with respect to that particular abstraction.

When we make an abstraction pattern-match against an abstraction involving scope specifiers, the behaviour should be as follows.

- Parts of (binding position) values annotated with neither **inner** nor **outer** are treated just as in previous versions of Fresh O’Caml: the atoms inside corresponding to free object-level names are collected and freshened. Such atoms therefore correspond to *binding* occurrences at the object language level.
- Parts of values annotated with **inner** do *not* have their atoms collected in this way; however, any atoms within that part of the value are treated as *bindable* occurrences of object-level names: ones that may be treated up to α -conversion. Such atoms may therefore be subject to freshening.
- Parts of values annotated with **outer** are just left alone on a pattern-match. Thus any atoms within them are treated as if they correspond to object-level names that are not to be treated up to α -conversion.

We hope that the introduction of these specifiers would help to increase the expressivity of Fresh O’Caml and believe that their implementation should be straightforward. One slight thorn in the side however, particularly when compared to Caml, is that the **inner** and **outer** type constructors would have to come along with associated data constructors (we call them **Inner** and **Outer**). These latter constructors would manifest themselves in patterns and value declarations: whether or not that is seen as a problem is really down to personal preference.

References

- [1] *The Objective Caml system*, available from <http://caml.inria.fr/>.
- [2] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Lèvy, *Explicit substitutions*, in: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California (1990)*, pp. 31–46.

- [3] Abramsky, S., D. Ghica, A. Murawski, L. Ong and I. D. B. Stark, *Nominal games and full abstraction for the nu-calculus*, in: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science* (2004), pp. 150–159.
- [4] Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich and S. Zdancewic, *Mechanized metatheory for the masses: The POPLmark challenge* (2005), submitted for publication.
- [5] Barendregt, H. P., “The Lambda Calculus: Its Syntax and Semantics,” North-Holland, 1984.
- [6] Benton, N. and B. Leperchey, *Relational reasoning in a nominal semantics for storage*, in: P. Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, Lecture Notes in Computer Science **3461** (2005), pp. 86–101.
- [7] Cardelli, L., J. C. Mitchell, S. Martini and A. Scedrov, *An extension of system F with subtyping*, *Information and Computation* **109** (1994), pp. 4–56.
- [8] Cheney, J., *Scrap your nameplate*, in: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP’05)* (2005).
- [9] Cheney, J. and C. Urban, *Alpha-prolog: A logic programming language with names, binding and alpha-equivalence*, in: B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, Lecture Notes in Computer Science **3132** (2004), pp. 269–283.
- [10] de Rauglaudre, D., *Camlp4 reference manual* (2004), available from <http://caml.inria.fr/camlp4/index.html>.
- [11] Gabbay, M. J., “A Theory of Inductive Definitions with α -Equivalence: Semantics, Implementation, Programming Language,” Ph.D. thesis, University of Cambridge (2000).
- [12] Gabbay, M. J. and A. M. Pitts, *A new approach to abstract syntax with variable binding*, *Formal Aspects of Computing* **13** (2002), pp. 341–363.
- [13] Hofmann, M., *Semantical analysis of higher-order abstract syntax*, in: *14th Logic in Computer Science Conference (LICS’99)* (1999), pp. 204–213.
- [14] Jech, T. J., “The axiom of choice,” North-Holland, 1973.
- [15] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation (PLDI)* (1988), pp. 199–208.
- [16] Pitts, A. M., *Nominal logic: A first order theory of names and binding*, in: N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001. Proceedings*, Lecture Notes in Computer Science **2215** (2001), pp. 219–242.

- [17] Pitts, A. M. and M. J. Gabbay, *A metalanguage for programming with bound names modulo renaming*, in: R. Backhouse and J. N. Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science **1837** (2000), pp. 230–255.
- [18] Pottier, F., *An overview of Cxml* (2005), submitted.
- [19] Sewell, P. E., J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit and V. Vafeiadis, *Acute: high-level programming language design for distributed computation. Design rationale and language definition* (2004), university of Cambridge Computer Laboratory Technical Report 605. Available from <http://www.cl.cam.ac.uk/users/pes20/acute/index.html>.
- [20] Shinwell, M. R., “The Fresh Approach: functional programming with names and binders,” Ph.D. thesis, University of Cambridge Computer Laboratory (2005).
- [21] Shinwell, M. R., A. M. Pitts and M. J. Gabbay, *FreshML: Programming with binders made simple*, in: *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP’03)* (2003), pp. 263–274.