An overview of Cαml

François Pottier [1]

INRIA

Abstract

Cαml is a tool that turns a so-called "binding specification" into an Objective Caml compilation unit. A binding specification resembles an algebraic data type declaration, but also includes information about names and binding. Cαml is meant to help writers of interpreters, compilers, or other programs-that-manipulate-programs deal with α-conversion in a safe and concise style. This paper presents an overview of Cαml's binding specification language and of the code that Cαml produces.

Key words: Metaprogramming, variable binding, α-conversion

1 Introduction

Functional programming languages such as ML and Haskell are intended for building, examining, and transforming complex symbolic objects, such as *programs* and *proofs*. They are well suited for this task because these objects are usually represented as *abstract syntax trees*, whose structure is easily expressed in ML or Haskell via algebraic data type declarations—or is it?

The whole truth is, abstract syntax trees involve *names* that can be *bound*. Manipulating such trees involves a number of operations that respect the meaning of names, such as computing the set of *free* names of a term, or substituting, *without capture*, a name (or term) for a name throughout a term. ML or Haskell provide no support for these operations. As a result, they have to be hand-coded using one of a variety of approaches, discussed below. This hand-coding process is tedious and error-prone. There clearly is a need for a more *declarative, robust, automated* approach to dealing with abstract syntax involving names and binding.

Three facets of the problem should perhaps be distinguished. First, one needs a *specification language* that provides a declarative way of describing the structure of abstract syntax trees, including binding information. Second,

[1] Email: `Francois.Pottier@inria.fr`

one needs an *implementation technique*, that is, an efficient runtime representation for names and binders. Last, there should be an *automated route* from specification to implementation. We now discuss these three points in turn.

◇ *Specification languages.* A number of specification languages have appeared in the literature. Plotkin [25] defines a notion of "binding signature" that allows for "function terms" of the form $(x_1, \ldots, x_n)\, t$, where the names $x_1, \ldots, x_n$ are considered bound within the term t. Talcott [33] introduces "binding structures", which are somewhat more expressive: each operator carries a fixed number of variables as well as a fixed number of sub-terms, and a fixed "binding relation" tells which variables are considered bound within which sub-terms. Honsell *et al.*'s "nominal algebras" [11] and Urban *et al.*'s "nominal signatures" [34] are essentially identical to Plotkin's binding signatures, except terms are typed, which in practice is important. Shinwell's Fresh Objective Caml [29,30,31] implements a generalized form of nominal signatures, where binding occurrences of names can inhabit a term of arbitrary structure, instead of only a tuple $(x_1, \ldots, x_n)$ of fixed size n. This, in particular, allows for constructs that bind a variable number of names.

Still, *we view none of these proposals as fully satisfactory:* there remain real-world constructs that cannot be faithfully modeled in any of these specification languages. In particular, constructs that bind a variable number of names, such as ML's let and let rec, are either not expressible at all or expressible only in a somewhat contorted way—for instance, by encoding a list of pairs as a pair of lists. Examples 2.3 and 2.4 provide details.

The first contribution of this paper is to propose a new *binding specification* language, which is sufficiently expressive to deal with these constructs and many others. An important aspect of this proposal is an expressive language of *patterns*. Patterns are used to form abstractions. Patterns are terms, so they can have arbitrary size. In particular, a pattern can bind a list of names of arbitrary length. Patterns can contain an arbitrary mixture of (binding occurrences of) names, sub-terms that lie *within* the scope of the abstraction, and sub-terms that lie *outside* the scope of the abstraction. A key point is that the *lexical* structure of a term (that is, the position of the binders and the extent of their scope) does not have to coincide with its *physical* structure.

◇ *Implementation techniques.* De Bruijn's encoding of abstract syntax trees with binding into standard first-order terms [7] is a well-known, and quite popular, implementation technique. It consists in encoding names (free or bound) as integer indices that can be interpreted as (relative) pointers to their binding site. Its strengths are that α-equivalence coincides with first-order equality and that a "fresh" name generator is not necessary. Its main disadvantage is that it makes the meaning of open terms context-dependent, requiring "shift" operations to compensate for changes in the context.

Another technique, used in the Fresh Objective Caml [29,30,31] and FreshLib [5] implementations, consists in encoding names as *atoms*—values that can

be tested for equality, but otherwise have no meaning. In this approach, when an abstraction is opened, the atoms that it binds have to be "freshened"—replaced with fresh atoms—in order to avoid name clashes. This technique is adopted here.

◇ *Automating implementation.* In practice, it is desirable to deal with binding issues by writing specifications, not code. In other words, there is a need for tools that can deal directly with binding specifications.

One way to satisfy this need is to incorporate this feature into a programming language design. For instance, Pitts and Gabbay's FreshML [24] is equipped with an *abstraction* type constructor. As a result, in FreshML, algebraic data type definitions are sufficiently expressive to encode nominal signatures in the style of Urban *et al.* [34]. Pattern matching against an abstraction automatically "freshens" all bound atoms. Fresh Objective Caml [29,30,31] is the successor to FreshML. Urban and Cheney's αProlog [6] is a variant of Prolog that is also able to encode nominal signatures. αProlog terms are unified up to α-equivalence.

Instead of designing a new programming language, however, it is also possible to write a separate tool that accepts a binding specification and produces code for an existing programming language. This is the approach followed in the present paper. It is less ambitious but simpler than that of Fresh Objective Caml, our most direct competitor. A practical benefit of simplicity is to make some useful optimizations, such as "lazy renaming", easier to implement. The actual design of such a tool is our second contribution.

Cheney's FreshLib [5] appears to be in the same spirit, but presents itself purely as a Haskell library. The need for a separate code generation tool is avoided by relying on Haskell's support for generic programming.

In this paper, we describe Cαml (pronounced: "alphaCaml") [26], a tool that accepts a flexible language of binding specifications and produces Objective Caml [12] code. (The tool could be easily adapted to other programming languages, such as Haskell.) We begin with a formal account of the syntax of (a simplified version of) the binding specification language (§2). We give meaning to this language by providing a formal definition of the α-equivalence relation that a binding specification gives rise to (§3). Then comes an informal description of the concrete binding specifications accepted by Cαml (§4) and of the code that is produced (§5). After a more complete comparison with related work (§6), we summarize our contribution and discuss directions for future work (§7).

2 Binding specifications

In this section and in the following one, we adopt a *structural* view of types and terms: that is, we build them out of a fixed, "universal" set of constructors. This is in contrast with the rather more common approach that consists in

$$
\begin{array}{lll}
s ::= \mathsf{inner} \mid \mathsf{outer} & & \textit{Scope specifiers} \\
\\
t ::= \mathsf{unit} \mid t \times t \mid t + t \mid \mathsf{atom} \mid \langle u \rangle & & \textit{Expression types} \\
u ::= \mathsf{unit} \mid u \times u \mid u + u \mid \mathsf{atom} \mid s\ t & & \textit{Pattern types} \\
\\
e ::= () \mid (e, e) \mid \mathsf{inj}_i\, e \mid a \mid \langle p \rangle & & \textit{Expressions} \\
p ::= () \mid (p, p) \mid \mathsf{inj}_i\, p \mid a \mid s\ e & & \textit{Patterns}
\end{array}
$$

Fig. 1. Types and terms

requiring a *signature* (or *specification*) to be first given. A signature typically specifies a set of (named) *type constructors*, each of which is equipped with a *sort*, giving rise to an algebra of types; and a set of (named) *term constructors*, each of which is equipped with an *arity*, giving rise to an algebra of terms. In practice, assigning names to type and term constructors is useful and desirable, and indeed our tool is driven by a programmer-supplied specification (§4). In this theoretical exposition, however, this level of detail is rather distracting, so we omit it altogether. Thus, the syntax of our types and terms is fixed, and appears in Figure 1.

Types are partitioned into *expression types*, written t, and *pattern types*, written u. Similarly, terms are partitioned into *expressions*, written e, and *patterns*, written p. We let a range over a countably infinite set $\mathbb{A}$ of *atoms*. We write $\mathit{atoms}(p)$ for the (finite) set of all atoms that appear (at any depth) within a pattern p, regardless of the intended meaning of their occurrences (free, bound, or binding).

Expressions and patterns can exhibit arbitrary structure. Indeed, both expression and pattern types include a unit type as well as binary products and sums. Accordingly, both expressions and patterns include a unit term, written $()$, pairs, and injections, written $\mathsf{inj}_i\, \cdot$, where i ranges over $\{1, 2\}$.

Both expression and pattern types include an atom type. Accordingly, both expressions and patterns include atoms a. The difference between expressions and patterns lies in the way these atoms are interpreted. In an expression, an occurrence of an atom is understood as a *reference* to some earlier occurrence of that atom in a binding position. In a pattern, an occurrence of an atom is understood as a *binding site* for that atom. For instance, when we encode the terms of the λ-calculus in our framework (Example 2.1), we will see that the leftmost occurrence of a inside (the encoding of) the term $(\lambda a.a)\, a$ lies inside a pattern, whereas the other two lie inside expressions.

Expressions include *abstractions* $\langle p \rangle$, where p is a pattern. Conversely, abstractions include an *end-of-abstraction* construct $s\ e$, where e is an expression and where the *scope specifier* s is one of inner and outer. Quite naturally, expression types and pattern types contain analogous constructs $\langle u \rangle$ and $s\ t$.

A pattern p can be thought of as a tree where every leaf carries either

$$ba(()) = ba(s\ e) = \varnothing \qquad ba((p_1, p_2)) = ba(p_1) \cup ba(p_2)$$
$$ba(\mathsf{inj}_i\ p) = ba(p) \qquad ba(a) = \{a\}$$

Fig. 2. The atoms bound by a pattern

an atom or a sub-expression decorated with a scope specifier. The set of all atoms that appear at some leaf of the former kind, written $ba(p)$, is referred to as the set of *atoms bound by* p. Leaves of the latter kind do not contribute to it. It is formally defined in Figure 2.

The meaning of abstractions can now be informally explained as follows. When an abstraction $\langle p \rangle$ is formed, the atoms bound by p become bound inside the sub-expressions of p that are decorated with inner. They do *not* become bound, however, inside sub-expressions that are decorated with outer. Thus, the inner and outer specifiers serve to distinguish between sub-expressions that lie inside the scope of the abstraction, on the one hand, and sub-expressions that happen to be physically attached to a leaf of the tree p, but do not lie inside the scope of the abstraction, on the other hand.

The correspondence between expressions and expression types, and between patterns and pattern types, is extremely straightforward. We omit its definition.

It is worth pointing out that the definition of types in Figure 1 can and should be viewed as co-inductive, giving rise to recursive types. This is necessary because families of data structures of unbounded size, such as lists and abstract syntax trees, only have recursive types. The definition of expressions and patterns, on the other hand, should be viewed as inductive: that is, expressions and patterns are finite terms. We do not wish to reason about α-equivalence of infinite terms, because this subtle notion has found little practical application so far.

The following two examples illustrate the intended meaning of our abstraction and end-of-abstraction constructs as well as the distinction between the scope specifiers inner and outer.

Example 2.1 The terms of the pure λ-calculus are given by the grammar

$$M := a \mid M\ M \mid \lambda a.M$$

with the proviso that, in a λ-abstraction $\lambda a.M$, the atom a is bound inside the λ-term M. In our framework, these terms are encoded as expressions of type t, where t is the unique solution to the following equation:

$$t = \mathsf{atom} + (t \times t) + \langle \mathsf{atom} \times \mathsf{inner}\ t \rangle$$

(For the sake of readability, our examples use n-ary products and sums when necessary.) The encoding of the λ-term $\lambda a.a$ is the expression of type t

$$\mathsf{inj}_3\ \langle (a, \mathsf{inner}\ \mathsf{inj}_1\ a) \rangle$$

According to the informal explanation above, the atom a is to be considered bound within this abstraction. Indeed, its leftmost occurrence causes it to be part of the atoms bound by the pattern $(a, \mathsf{inner}\ \ldots)$. Its scope is the sub-expression $\mathsf{inj}_1\, a$, which is decorated with inner. Thus, the whole expression should be considered α-equivalent to

$$\mathsf{inj}_3\, \langle (a', \mathsf{inner}\ \mathsf{inj}_1\, a') \rangle$$

which encodes the λ-term $\lambda a'.a'$. Our definition of α-equivalence (§3) indeed relates these expressions.

Example 2.2 The pure λ-calculus is often extended with a let construct for local definitions:

$$M := \ldots \mid \mathsf{let}\ a = M\ \mathsf{in}\ M$$

In a local definition $\mathsf{let}\ a = M_1\ \mathsf{in}\ M_2$, the atom a is bound inside M_2. In our framework, λ-terms can then be encoded as expressions of type t, where t satisfies

$$t = \ldots + \langle \mathsf{atom} \times \mathsf{outer}\ t \times \mathsf{inner}\ t \rangle$$

The encoding of the λ-term $\mathsf{let}\ a = a\ \mathsf{in}\ a$ is the expression

$$\mathsf{inj}_4\, \langle (a, \mathsf{outer}\ \mathsf{inj}_1\, a, \mathsf{inner}\ \mathsf{inj}_1\, a) \rangle$$

Again, the atom a is bound in this abstraction, and its scope consists of the inner sub-expression alone—the outer sub-expression does not lie in its scope. Thus, this expression should be considered α-equivalent to

$$\mathsf{inj}_4\, \langle (a', \mathsf{outer}\ \mathsf{inj}_1\, a, \mathsf{inner}\ \mathsf{inj}_1\, a') \rangle$$

which encodes the λ-term $\mathsf{let}\ a' = a\ \mathsf{in}\ a'$.

The previous two examples illustrate the use of inner and outer. Still, it is perhaps not clear yet why outer is useful at all, and why patterns are allowed to contain multiple sub-expressions. Let us, *a contrario*, temporarily consider a restricted language where outer is suppressed and where every abstraction contains a single, distinguished sub-expression, implicitly decorated with inner. The production $p ::= s\ e$ is suppressed, so that patterns no longer contain expressions, and the unary abstraction construct $\langle p \rangle$ is replaced with a binary one, of the form $\langle p \rangle e$, where the atoms in $ba(p)$ are considered bound inside e. Analogous changes are made at the level of types.

It is clear that this restricted language can be embedded within the full language. Indeed, $\langle p \rangle e$ can be encoded as $\langle (p, \mathsf{inner}\ e) \rangle$, and $\langle u \rangle t$ as $\langle u \times \mathsf{inner}\ t \rangle$. So, the latter is at least as expressive as the former.

The converse, however, is true only to a certain extent. Certainly, the restricted language is quite expressive. It subsumes nominal algebras [11] as well as nominal signatures [34], and corresponds to a fragment of Fresh Objective Caml [29,31,30]. In particular, it is expressive enough to adequately deal with

Examples 2.1 and 2.2. Indeed, the type $\langle \mathsf{atom} \times \mathsf{inner}\ t \rangle$ of Example 2.1 can be written $\langle \mathsf{atom} \rangle\, t$ in the restricted language. The type $\langle \mathsf{atom} \times \mathsf{outer}\ t \times \mathsf{inner}\ t \rangle$ of Example 2.2 can be written $t \times \langle \mathsf{atom} \rangle\, t$, if one is willing to alter the order in which the three components are listed and to hoist $\mathsf{outer}\ t$ outside of the abstraction, where it simply becomes t. Yet, there are situations where no such trick can be pulled. These typically involve patterns that bind a variable number of atoms. We claim that, in these cases, our more general language offers superior flexibility. The next two examples substantiate this claim.

Example 2.3 Realistic extensions of the λ-calculus allow local definitions to bind several atoms at once:

$$M ::= \ldots \mid \mathsf{let}\ a = M\ \mathsf{and}\ \ldots\ \mathsf{and}\ a = M\ \mathsf{in}\ M$$

In a definition $\mathsf{let}\ a_1 = M_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ a_n = M_n\ \mathsf{in}\ M$, the atoms $\{a_1, \ldots, a_n\}$ are bound in M. They are not bound in $M_1, \ldots, M_n$. In our framework, such λ-terms are encoded as expressions of type t, where t and the auxiliary pattern type u satisfy the equations

$$t = \ldots + \langle u \times \mathsf{inner}\ t \rangle$$
$$u = \mathsf{unit} + \mathsf{atom} \times (\mathsf{outer}\ t) \times u$$

The second equation defines u as the type of lists of pairs of an atom and an outer expression. Thus, a list of bindings $a_1 = M_1, \ldots, a_n = M_n$ can be encoded as a pattern of type u. The first equation states that a let definition is encoded as an abstraction that consists of such a list of bindings and of an inner expression, which encodes the final λ-term M. By definition (Figure 2), the atoms bound by this abstraction are the atoms $\{a_1, \ldots, a_n\}$ that appear in the list of bindings, and their scope is the inner expression that encodes M. Their scope does not encompass the expressions that encode $M_1, \ldots, M_n$, even though these expressions physically lie inside the abstraction, because they are marked outer.

How does one deal with this situation in the restricted formalism that was discussed above? The (encodings of the) terms $M_1, \ldots, M_n$ must lie out of the scope of the abstraction, so, in the restricted language, they must physically lie out of the abstraction. The bound atoms $a_1, \ldots, a_n$, on the other hand, must lie inside the left-hand side of the abstraction. As a result, one must maintain two separate lists: a list of expressions and a list of atoms.

$$t = \ldots + t' \times \langle u \rangle t$$
$$t' = \mathsf{unit} + t \times t'$$
$$u = \mathsf{unit} + \mathsf{atom} \times u$$

This encoding is awkward and fragile. Indeed, it introduces *junk*: one can

accidentally break the property that the two lists have the same length and construct a term that does not encode a valid λ-term. By contrast, in our new formalism, the fact that our patterns can have an unbounded number of sub-expressions, instead of just one, and the fact that these sub-expressions can be marked inner or outer, provide enough flexibility to resolve this issue, which Shinwell [29, pages 19–20] identifies but leaves open.

Example 2.4 Let us now extend the λ-calculus with *mutually recursive* local definitions:

$$M ::= \ldots \mid \mathsf{letrec}\ a = M\ \mathsf{and}\ \ldots\ \mathsf{and}\ a = M\ \mathsf{in}\ M$$

In a definition $\mathsf{letrec}\ a_1 = M_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ a_n = M_n\ \mathsf{in}\ M$, the atoms $\{a_1, \ldots, a_n\}$ are bound in $M_1, \ldots, M_n$ and in M. In our framework, such λ-terms are encoded as expressions of type t, where t and the auxiliary pattern type u satisfy the equations

$$t = \ldots + \langle u \times \mathsf{inner}\ t \rangle$$
$$u = \mathsf{unit} + \mathsf{atom} \times (\mathsf{inner}\ t) \times u$$

This definition is almost identical to that of Example 2.3. The only difference is that the equation that defines the pattern type u now mentions inner t instead of outer t. Thus, the scope of the atoms bound by the pattern contains not only (the encoding of) the λ-term M, but also (the encodings of) the λ-terms $M_1, \ldots, M_n$ that appear in the list of bindings. Again, we achieve this effect without splitting the list of bindings into two separate lists, whereas previous approaches appear unable to do so. In particular, the restricted formalism that was discussed above requires separating the list of atoms $a_1, \ldots, a_n$, which must lie in the left-hand side of the abstraction, and the list of expressions that encode $M_1, \ldots, M_n$, which must lie in the right-hand side of the abstraction [29, page 20].

Abstractions in Cαml intentionally cannot be directly nested: when an abstraction is opened, it must be closed (via inner or outer) before another abstraction is introduced. This design choice makes it easier to define and understand the meaning of inner and outer. Fresh Objective Caml [29,30,31] is more liberal and allows left-nesting of (binary) abstractions, which currently has no analogue in Cαml.

3 A definition of α-equivalence

We now give precise meaning to the language introduced in the previous section by formally defining when two terms are α-equivalent. Our definition of α-equivalence is in the style of Pitts [23], but requires a few preliminary definitions in order to deal with our rich language of patterns.

$$
\begin{array}{rlrl}
[a_1/a_2] &= \{(a_2, a_1)\} & [\mathsf{inj}_i\, p_1/\, \mathsf{inj}_i\, p_2] &= [p_1/p_2] \\
[()/()] &= \varnothing & [(p_1, p_1')/(p_2, p_2')] &= [p_1/p_2] \cup [p_1'/p_2'] \\
[s\ e_1/s\ e_2] &= \varnothing & & \text{if this relation is a renaming}
\end{array}
$$

Fig. 3. Relating two patterns via a renaming

3.1 Renamings

Definition 3.1 A *renaming* is a *finite, bijective* mapping of atoms to atoms, that is, a bijection between two finite subsets of $\mathbb{A}$. A renaming is implicitly viewed as a total (but not necessarily bijective) mapping of atoms to atoms, of patterns to patterns, and of expressions to expressions.

A renaming is viewed as a mapping of patterns to patterns and of expressions to expressions in the most straightforward way. That is, every occurrence of an atom within a term is renamed. Whether this occurrence is meant to bind a new name or to refer to an existing name is disregarded. No precautions against capture are taken.

Standard treatments of α-equivalence, such as Pitts' [23], rely on singleton renamings of the form $\{(a_2, a_1)\}$, where a_1 and a_2 are atoms. Such a renaming, usually written $[a_1/a_2]$, maps a_2 to a_1. Applying it to an expression e blindly replaces every occurrence of a_2 within e with a_1. Capture is usually avoided by requiring the atom a_1 to be fresh for e.

In this paper, we need to construct more complex renamings, which we write $[p_1/p_2]$, where p_1 and p_2 are patterns of a common type. Such a renaming is defined only when p_1 and p_2 have identical "pattern structure", that is, when they differ only in (i) the identity of their bound atoms and (ii) their (inner or outer) sub-expressions.

Definition 3.2 The partial function $[\cdot/\cdot]$, defined in Figure 3, maps pairs of patterns of a common type to renamings. That is, if p_1 and p_2 are two patterns of a common type u, then $[p_1/p_2]$ is either undefined or a renaming. When it is defined, its domain is $ba(p_2)$ and its range is $ba(p_1)$.

The upper left equation in Figure 3 states that $[a_1/a_2]$ is the singleton renaming that maps a_2 to a_1.

The next equation down states that the empty renaming relates the unit pattern () with itself.

The bottom left equation states that sub-expressions are ignored in the construction of a renaming between patterns. It does not matter if they differ.

The top right equation states that $[\mathsf{inj}_i\, p_1/\, \mathsf{inj}_j\, p_2]$ is undefined when i and j are distinct (and, of course, when $[p_1/p_2]$ is undefined). That is, only patterns that have identical structure can be related. In the case of constructs that bind a variable number of atoms, such as those presented in Examples 2.3 and 2.4,

$$()^s = () \qquad (p_1, p_2)^s = (p_1^s, p_2^s) \qquad (s\ e)^s = e$$
$$a^s = () \qquad (\mathsf{inj}_i\, p)^s = \mathsf{inj}_i\, p^s \qquad (s'\ e)^s = () \quad \text{if } s \neq s'$$

Fig. 4. Collecting the inner or outer sub-expressions of a pattern

$$() =_\alpha () \qquad \frac{e_1 =_\alpha e_2 \qquad e_1' =_\alpha e_2'}{(e_1, e_1') =_\alpha (e_2, e_2')} \qquad \frac{e_1 =_\alpha e_2}{\mathsf{inj}_i\, e_1 =_\alpha \mathsf{inj}_i\, e_2} \qquad a =_\alpha a$$

$$\frac{p_1^{\mathsf{outer}} =_\alpha p_2^{\mathsf{outer}} \qquad [p/p_1]\, p_1^{\mathsf{inner}} =_\alpha [p/p_2]\, p_2^{\mathsf{inner}} \qquad \mathit{atoms}(p) \cap \mathit{atoms}(p_1, p_2) = \varnothing}{\langle p_1 \rangle =_\alpha \langle p_2 \rangle}$$

Fig. 5. α-equivalence of expressions

this means, in particular, that only patterns that bind the same number of atoms can be related.

The last equation states that, in order to relate two pair patterns (p_1, p_1') and (p_2, p_2'), one decomposes the problem component-wise. That is, one builds the renamings $[p_1/p_2]$ and $[p_1'/p_2']$, forms their set-theoretic union, which yields a relation on atoms, and checks that this relation is a renaming. This check fails if the new relation is not applicative or if it is not injective. This occurs, for instance, when attempting to evaluate $[(x, y)/(z, z)]$ or $[(z, z)/(x, y)]$ for distinct atoms x, y, z. The checks succeed, on the other hand, when evaluating $[((x, y), x)/((z, x), z)]$. This yields the renaming that maps z to x and x to y. Note that patterns need not be linear: a single atom can have multiple occurrences in a pattern. Our definition only requires that all occurrences be consistently renamed.

3.2 α-equivalence

Definition 3.3 If p is a pattern and s is a scope specifier, then p^s denotes an expression, defined in Figure 4.

This transformation erases all atoms within p, as well as all sub-expressions *not* marked s, by replacing them with $()$. In short, p^s can be viewed as a collection of the sub-expressions marked s in p. The structure of the pattern (pairs, injections) is preserved, but will be irrelevant.

Definition 3.4 α-equivalence over expressions of a common type is defined by the rules of Figure 5.

The definition is by induction on the size of expressions, as opposed to structural induction, because of the pruning and renaming performed in the last rule.

The first four rules in Figure 5 are standard congruence rules. We focus on the last rule, which specifies when two abstractions $\langle p_1 \rangle$ and $\langle p_2 \rangle$ are α-equivalent.

The first premise checks that any sub-expressions that lie in outer scope are α-equivalent. The auxiliary function $(\cdot)^{\mathsf{outer}}$ erases all atoms in binding position as well as all inner sub-expressions, so that they do not participate in this check.

The second premise requires $[p/p_1]$ and $[p/p_2]$ to be defined for a certain pattern p. This means that p_1 and p_2 have the same structure (ignoring their sub-expressions) and differ only up to a consistent renaming of the atoms that they bind. In order to overcome this difference, one maps them both to a common pattern p: that is, one checks that, under the renamings $[p/p_1]$ and $[p/p_2]$, the inner sub-expressions of p_1 and p_2 are α-equivalent. These sub-expressions are collected using the auxiliary function $(\cdot)^{\mathsf{inner}}$.

The last premise requires p to be chosen fresh for p_i, where i ranges over $\{1, 2\}$. This allows viewing $[p/p_i]$, a bijection of domain $ba(p_i)$, as a bijection of domain $atoms(p_i)$. This prevents capture, that is, confusion between bound and free atoms, inside p_i.

When p_1 and p_2 have type $\mathsf{atom} \times \mathsf{inner}\ t$, the rule simplifies down to

$$\frac{[a/a_1]\, e_1 =_\alpha [a/a_2]\, e_2 \qquad a \notin atoms(a_1, e_1, a_2, e_2)}{\langle (a_1, \mathsf{inner}\ e_1) \rangle =_\alpha \langle (a_2, \mathsf{inner}\ e_2) \rangle}$$

Up to the differences in notation, this is exactly the definition of α-equivalence found, for instance, in Pitts' work [23].

Pitts' proof that α-equivalence is indeed an equivalence relation [22] is easily extended to our setting:

Theorem 3.5 $=_\alpha$ *is an equivalence relation.*

4 Concrete specifications

The previous sections have given a simplified account of our type and term languages. The binding specifications accepted by Cαml are more complex in several ways.

Figure 6 shows a binding specification, in concrete syntax, for an untyped λ-calculus. The most obvious difference with respect to the theoretical presentation is that entities are *named*. Indeed, the specification defines a collection of named *types* (*expression*, *lambda*, etc.). In the tradition of algebraic data type definitions, each type is defined either as a sum type or as a product type. Each sum type comes with a set of named *data constructors* (*EVar*, *ELambda*, etc.). In Figure 6, all product types are tuple types, whose fields are anonymous. Cαml also offers record types, whose fields are named. All definitions are mutually recursive: their scope is the entire specification. In

```
sort var

type expression =
  | EVar of atom var
  | ELambda of < lambda >
  | EApp of expression * expression
  | EPair of expression * expression
  | EInj of [ int ] * expression
  | ECase of expression * branch list
  | ELetRec of < letrec >

type lambda binds var =
    atom var * inner expression

type branch =
    < clause >

type clause binds var =
    pattern * inner expression

type letrec binds var =
    binding list * inner expression

type binding binds var =
    pattern * inner expression

type pattern binds var =
  | PWildcard
  | PVar of atom var
  | PPair of pattern * pattern
  | PInj of [ int ] * pattern
  | PAnd of pattern * pattern
  | POr of pattern * pattern
```

Fig. 6. Concrete binding specification for an untyped λ-calculus

short, Cαml offers *iso-recursive* types, whereas, for simplicity, the theoretical account in §2–3 is based on *equi-recursive* types.

Cαml is able to deal with multiple *sorts* of atoms. In Figure 6, only one such sort (*var*) is declared. Every occurrence of the atom keyword is followed with a sort. A specification that involves multiple sorts is shown further on.

In concrete syntax, expression types and pattern types are distinguished by the fact that the latter (and only the latter) carry a binds clause. Such a clause consists of the binds keyword, followed by one or several sorts. In Figure 6, the clause "binds *var*" distinguishes the pattern types. An occurrence of "atom *var*" inside the definition of such a type is understood as a binding occurrence.

The definitions of the data constructor *ELambda* and of the type *lambda* follow Example 2.1. The definitions of the data constructor *ELetRec* and of the types *letrec* and *binding* follow Example 2.4. Furthermore, the sample specification includes binary products and sums (*EPair*, *EInj*), a case construct (*ECase*), and a rich language of patterns (*pattern*). Patterns are used both in case constructs and in let rec definitions. (The definitions of the types *binding* and *clause* happen to be identical. We keep them separate for clarity.) A branch is defined as an abstraction over a pair of a pattern and an inner expression. There, every atom that appears in the left-hand pattern is considered bound in the right-hand expression. A let rec definition is defined as an abstraction over a pair of a list of bindings and an inner expression. There, every atom that appears in the left-hand pattern of *some* binding is considered bound in the right-hand expression of *every* binding *and* in the right-hand expression of the let rec construct. This provides a good illustration of the flexibility of the specification language.

A few more features are worth discussing:

```
sort typevar

type typ =
  | TVar of atom typevar
  | TArrow of typ * typ
  | ...
```

```
type expression =
  | ...
  | ETypeAnnotation of expression * typ

type pattern binds var =
  | ...
  | PTypeAnnotation of pattern * neutral typ
```

Fig. 7. Concrete binding specification for a typed λ-calculus (excerpts)

◇ *Containers.* The type constructor *list*, used in the definition of *letrec*, comes from Objective Caml's standard library. It is known to Cαml by default. The type constructor *option* is also built-in. Other "container" types can be used via a dedicated uses container declaration, which mentions not only the name of the type constructor, but also those of its *map* and *fold* functions. For instance, if *list* was not known to Cαml, it could be declared as follows:

uses container *list* with *List.map* and *List.fold_left*

◇ *Linearity.* Here, the object language includes conjunction and disjunction patterns (*PAnd*, *POr*). One might want to check that the two sides of a conjunction pattern have disjoint sets of bound atoms, and that the two sides of a disjunction pattern have identical sets of bound atoms. Cαml does not need to know (and, in fact, cannot be told) about this well-formedness condition. It is up to the user to enforce it. This is acceptable, since this condition does not interact with α-conversion issues in any way.

◇ *Objective Caml escape hatch.* The data constructors for injections (*EInj*, *PInj*) carry an integer parameter, whose type is the Objective Caml type *int*. Arbitrary Objective Caml type expressions are allowed in specifications when surrounded with square brackets. Inside square brackets, α-equivalence is the identity. In other words, Cαml pretends that the values inside square brackets never contain atoms: it never attempts to scan them for bound atoms or to rename their free atoms. This is useful, in particular, when one wishes to attach mutable information to terms. Mutable record fields or ref cells are not directly permitted in specifications, but are allowed inside square brackets.

◇ *Multiple sorts of atoms.* Figure 7 sketches how to enrich the untyped λ-calculus of Figure 6 with type annotations that can appear inside object-level expressions and patterns. Type variables are represented by a new sort of atoms (*typevar*), so they are considered distinct from term variables. Types contain atoms of sort *typevar*. Expressions can now contain type annotations, via a new data constructor *ETypeAnnotation*, so expressions now contain atoms of sorts *var* and *typevar*. Similarly, patterns can now contain type annotations, via a new data constructor *PTypeAnnotation*, so patterns now contain atoms of sorts *var* and *typevar*, but bind only the former.

In the definition of *PTypeAnnotation*, the second parameter, whose type

```
type expression =
  | EVar of var
  | ELambda of lambda
  | EApp of expression * expression
  | EPair of expression * expression
  | EInj of ( int ) * expression
  | ECase of expression * clause list
  | ELetRec of letrec

and var =
  Identifier.t

and lambda =
  var * expression
```

Fig. 8. Raw version of the specification of Figure 6 (excerpt)

is *typ*, is preceded with the keyword neutral, whereas, according to what we have said so far, one would expect inner or outer. Indeed, in this particular situation, the distinction between inner and outer becomes pointless: *pattern* binds atoms of sort *var*, which *typ* never refers to, so whether *typ* lies inside or outside the scope of the abstraction makes no difference. In order to spare the user an arbitrary decision between inner and outer, we provide the keyword neutral. The use of this keyword is permitted and required where inner and outer would be equivalent. This provides a useful sanity check.

5 From specification to code

Out of a binding specification, Cαml produces Objective Caml type definitions and code. Within type definitions, all Cαml-specific markup is eliminated. Some of it, such as inner, outer, and neutral keywords, binds clauses, and square brackets, is simply erased. Some other aspects, including occurrences of the atom keyword and abstractions, is dealt with in two distinct ways. That is, Cαml produces *two* versions of each type definition, a *raw* version and an *internal* one.

5.1 "Raw" type definitions

Figure 8 shows a fragment of the raw version of the specification of Figure 6. In this version, all occurrences of atom are translated down to the Objective Caml type *Identifier.t*, which by default is synonymous with *string*. An alternative definition of *Identifier* can be supplied if desired; for instance, identifiers could be pairs of a string and of an offset into some source file. Abstractions are erased.

The raw version is intended only for conversion to and from textual form, that is, for production by a parser and for consumption by a pretty-printer. Functions that convert back and forth between raw and internal forms are automatically produced by Cαml. On the way in (from raw to internal form), they check that every identifier is correctly bound, and turn identifiers into an internal representation of atoms. On the way out, they turn atoms back into human-readable identifiers. The fact that parsers and pretty-printers need

```
type expression =                          and var =
  | EVar of var                              Var.Atom.t
  | ELambda of opaque_lambda
  | EApp of expression * expression        and lambda =
  | EPair of expression * expression          var * expression
  | EInj of ( int ) * expression
  | ECase of expression * clause list      and opaque_lambda
  | ELetRec of opaque_letrec

val create_lambda : lambda → opaque_lambda
val open_lambda : opaque_lambda → lambda
```

Fig. 9. Internal version of the specification of Figure 6 (excerpt)

only deal with raw forms means that they can be written in a standard style, without worrying about names and binding.

5.2 "Internal" type definitions

Figure 9 presents a fragment of the internal version of the specification of Figure 6.

In this version, each sort of atom is translated down to a distinct abstract type. More precisely, "atom *var*" is translated down to *Var.Atom.t*, where the module *Var* defines an abstract type of atoms, equipped with a number of operations. Similarly, "atom *typevar*" is translated to *Typevar.Atom.t*. The modules *Var* and *Typevar* have identical signatures, but define distinct abstract types, so the user cannot mistakenly confuse object-level type and term variables.

Each abstraction gives rise to *two* types, one of which is abstract, one of which is transparent. For instance, the type *lambda* of Figure 6 gives rise, in Figure 9, to the abstract type *opaque_lambda*, whose actual definition is not made public, and to the transparent type *lambda*, which is synonymous for a pair of an atom and an expression. Abstractions are translated down to their abstract versions: for instance, in Figure 9, the data constructor *ELambda* carries a parameter of type *opaque_lambda*.

Because *opaque_lambda* is an abstract type, a value of this type is not directly usable. The only way of exploiting it is for the programmer to explicitly invoke the function *open_lambda*, also produced by Cαml, which turns it into a transparent form. Whenever it is invoked, *open_lambda* produces a *fresh* atom a and returns a transparent copy of the abstraction where the bound atom has been consistently replaced with a. This semantics is identical to that of Fresh Objective Caml [32]. It automatically enforces Barendregt's convention that "the bound variables occurring in a certain expression are different from the free ones" [2]. Conversely, in order to turn a pair of an atom and

an expression into a value of type *opaque_lambda*, one must go through the function *create_lambda*.

5.3 More code

Experience suggests that automatically "freshening" the bound atoms upon opening abstractions obviates the need for many explicit renaming operations. Nevertheless, there remain situations where it is necessary to explicitly deal with names and renamings. Cαml automatically produces code for computing the sets of free or bound atoms of a term and for applying a substitution (of atoms for atoms) to a term.

Cαml also produces object-oriented code that follows the classic *visitor* design pattern [4] and helps succinctly define *transformations* and *traversals* over terms. A generated class called *map* provides a collection of methods (one per type, data constructor, or record field that appears in the binding specification). Each of these methods returns a deep copy of its argument, and is implemented via self-calls to other methods of the class. This makes it particularly easy to define transformations that behave "almost like" the identity. For instance, in the case of the language specified in Figure 6, capture-avoiding substitution of expressions for variables is implemented by overriding a single method, namely the one that deals with the data constructor *EVar*. This requires less than ten lines of user-written code, regardless of the size of the specification. A generated class called *fold* provides a similar facility for defining traversals.

5.4 Comments

Atoms are internally represented as pairs of an integer and an identifier. The integer alone represents the atom's identity; the identifier is used only as a hint in the conversion from internal forms back to raw forms, where atoms must be converted back to identifiers. The integer identity of an atom is accessible when required. Sets of atoms and maps over atoms are represented as Patricia trees [18] for performance.

Substitutions of atoms for atoms are applied eagerly at all nodes, except at abstraction nodes, where they are suspended until the abstraction is opened. This lazy approach allows saving work if the abstraction is never opened. More importantly, it allows multiple substitutions, including the "freshening" substitution that is required upon opening an abstraction, to be composed. This helps avoid successive applications of multiple substitutions to a single term. In particular, traversing a term, opening abstractions as they are encountered, only requires linear time under this lazy approach. In comparison, an eager approach, as currently implemented in Fresh Objective Caml [29,30,31], leads to quadratic time complexity. Of course, the idea of suspending substitutions is not new: consult, for instance, Nadathur and Qi [16] or Shinwell [29, page 162].

A global integer counter is used to produce fresh atoms when required. This is the only piece of global state maintained by the generated code. It is expected that, at any time, the value of the counter is greater than that of any *free* atom in existence. It is perfectly fine, however, if the value of the counter happens to be less than that of an atom that occurs *bound* inside a term. Indeed, the only way of getting hold of such an atom would be to open the abstraction that encloses it, but doing so causes it to be replaced with a fresh atom, so there is really no way of observing it. This remark has an important practical consequence: it implies that it is fine to apply *output_value* and *input_value*, Objective Caml's primitive operations for marshaling and unmarshaling, to a *closed* term—one that has no free atoms.

Cαml provides no support for dealing with names that are not explicitly bound, such as record labels and module names in Objective Caml, or package and class names in Java. These names are not subject to α-conversion, so the only problem to be solved is one of efficiency: for instance, one might wish to represent them as integers internally. This is easily hand-coded using either global state (which breaks compatibility with *output_value* and *input_value*) or a fixed hashing scheme in the style of Garrigue [8].

6 Related work

6.1 Specification languages

Many of the specification languages found in the literature [25,11,34,29] are less expressive than the one presented in this paper. In particular, they either cannot deal at all or cannot elegantly deal with object-level constructs, such as let and letrec, that bind a variable number of names.

The same limitation exists in Talcott's "binding structures" [33], where each construct must bind a fixed number of names. Yet, Talcott's language is in some aspects more expressive than the one presented here. Indeed, its "binding relations" allow specifying exactly which subset of the bound atoms is in scope within each sub-expression of an abstraction. This offers some extra flexibility that we have not deemed necessary. Here, inner encodes the full set of bound atoms, outer encodes the empty set, and no other set can be expressed.

The latest version of Fresh Objective Caml [30] offers *restricted abstraction* types, of the form $\langle t_1 \mid s \rangle\, t_2$, where t_1 and t_2 are types and s is a sort. The idea is that all atoms of sort s that occur inside the left-hand member are considered bound within the right-hand member. Thus, which sorts of atoms are considered bound is specified *a posteriori*, after t_1 is defined. In contrast, Cαml's specification language requires this decision to be made *a priori* and reflected in the binds clause for t_1. Either approach has its merits. Ours is less flexible and perhaps more verbose. It was deemed that requiring every pattern type to carry an explicit binds clause would, in the end, make specifications

more readable.

Cheney [5] describes the design of FreshLib, a Haskell library whose aim and motivations are analogous to those of Cαml. Like Fresh Objective Caml, FreshLib offers binary abstractions. Furthermore, it allows specifying, via hand-coded instance declarations, which atoms inside the pattern should be considered bound. This apparently offers good expressiveness, but in a style that is not fully declarative.

Higher-order abstract syntax [20] consists in encoding object-level abstractions as meta-level abstractions. In so doing, one shifts the burden of dealing with α-conversion onto the meta-language implementor. This approach has received extensive use in systems such as λ-Prolog or Twelf. Nevertheless, its use in the setting of functional programming languages appears awkward and has remained marginal. Indeed, as argued by Sheard [28, Section 13], encoding an object-level abstraction as *some kind of* meta-level abstraction is fine, but encoding it as a meta-level *lambda*-abstraction is not, because the *semantics* of λ-abstraction gets in the way: λ-abstractions do not support pattern matching, delay computational effects, introduce "junk" terms, etc. To avoid these problems, several authors [15,19,27] have studied extensions of a functional programming language with a new meta-level abstraction, whose only role is to permit higher-order abstract syntax encodings.

6.2 Implementation techniques

De Bruijn's encoding of (free and bound) names as integer indices [7] remains popular, because it is, in principle, easy to understand and to implement. However, this encoding is unsafe. Indeed, because names are integers, they can be forged: arbitrary integer values can be silently coerced into (meaningless) names. Also, the meaning of a de Bruijn index is relative to a context: when the context grows, because a new abstraction is entered, all existing names must be shifted up. Forgetting to do so, or failing to do so in a correct manner, has the effect of silently changing a name into another name. Typechecking techniques based on nested data types [3] or on generalized algebraic data types [36] can be used to ensure that all indices are within bounds, but it is not yet clear whether these techniques are practical. Designing a version of Cαml based on de Bruijn indices, without giving up any of its current safety guarantees, would require more research.

By contrast, in FreshML [24,32], names are unforgeable atoms, whose meaning is context-independent. Fresh names are automatically produced when an abstraction is opened, so "forgetting to shift" is impossible. Our approach is in the spirit of FreshML and shares these advantages.

The most recent and practical implementation of FreshML is Fresh Objective Caml [29,30,31], an extension of the Objective Caml compiler and runtime system. This very interesting piece of work was our most direct source of inspiration. Each of the Fresh Objective Caml and Cαml approaches has its

own set of advantages and disadvantages.

In the Fresh Objective Caml approach, it is difficult to efficiently implement the primitive operation that swaps two atoms (or renames an atom) throughout a term. This operation does not have access to types, because they are erased before execution. Instead, runtime checks must be used to tell which blocks represent atoms, abstractions, or ordinary terms. A runtime check is also necessary, in principle, to tell which blocks represent reference cells—within which no swapping should take place, according to Fresh Objective Caml's dynamic semantics. Unfortunately, Objective Caml's runtime system does not maintain this information, so this check currently cannot be implemented. Last, runtime checks are required when collecting the bound atoms of a restricted abstraction—whenever an atom is found, its sort must be compared with the sort that restricts the abstraction.

In contrast, the renaming code produced by Cαml contains no such runtime checks. Indeed, code generation in Cαml is type-directed, so it is statically known where to look for free atoms of a certain sort, where to look for bound atoms of a certain sort, and where to stop looking—for instance, under square brackets, or in sub-terms that are known not to contain any atoms of the desired sort. The ref type can be used only under square brackets, so reference cells are never traversed at runtime.

Because Fresh Objective Caml is a modification of the Objective Caml compiler and runtime system, as opposed to a separate tool or library, keeping it up-to-date requires more work.

Fresh Objective Caml is more ambitious than Cαml in its treatment of sharing and of cyclic terms. In Fresh Objective Caml, cyclic terms are supported and, when possible, sharing is preserved by renamings. The necessary algorithms are quite subtle and require care. In Cαml, on the other hand, sharing is ignored, so it is lost when applying renamings. Furthermore, terms are assumed not to be cyclic[2]. This is consistent with our formalization in §2, where expressions and patterns are inductively defined terms.

Cαml supports parameterized data type definitions, with a strong restriction: type variables cannot stand for Cαml expression types or pattern types. Instead, they must stand for plain Objective Caml types. Indeed, it was deemed difficult to design (and to efficiently compile) a specification language where expression and pattern types can be variables. Fresh Objective Caml has no such restriction.

McBride and McKinna [14] describe an implementation technique where free variables are represented by names and bound variables by de Bruijn indices. This offers the advantage, compared with de Bruijn's original approach, that the meaning of terms is again context-independent. In other words, en-

[2] Since ref types are disallowed in binding specifications, references cannot be exploited to create cyclic data structures. The only remaining way of creating such structures is via Objective Caml's liberal let rec construct. The Cαml user is warned against using this feature. Unfortunately, there is no way of actually prohibiting its use.

tering a new abstraction does not require existing terms to be shifted. Instead, a de Bruijn index is turned into a name, which must be chosen fresh.

This mixed approach apparently does not differ very much from a fully nominal, FreshML-style approach. The main advantage of representing bound variables as de Bruijn indices, as opposed to names, is that α-equivalence then coincides with standard, first-order equality. This makes it possible to use the equality and hashing operations provided by the host programming language, instead of generating code for these operations. One disadvantage is that creating an abstraction becomes just as costly as opening one, whereas it is a constant-time operation in the purely name-based approach. If desired, Cαml could probably be modified to follow this approach without any user-visible changes.

In fact, McBride and McKinna's approach is original in that names are not atoms. Instead, names have hierarchical structure. This allows picking fresh names in a local, deterministic manner, instead of relying on a global name generator. It is not clear how much of an advantage this is. A disadvantage, on the other hand, is that name comparison becomes significantly more costly.

FreshLib [5] is purely a library: no separate code generator is required. This surprising feat is achieved thanks to the Glasgow Haskell compiler's support for generic programming, that is, type-directed code generation [9]. Relying on a separate code generator, as we do, is in comparison low technology, but offers more freedom—such as the freedom to design our own specification language. Cheney exploits Haskell-specific techniques [13] in order to implement *modular generic traversals*, which in turn allow easily defining transformations such as capture-avoiding substitution. Cαml, instead, relies on Objective Caml's object-oriented features and on the classic *visitor* design pattern to define such traversals in a straightforward way.

Distinguishing raw and internal forms, and automatically producing code that converts back and forth between the two, appears to be a unique feature of Cαml. This certainly saves a lot of boilerplate code. Yet, some experience is needed to tell whether this feature is useful in real-world situations.

7 Conclusion

7.1 Contributions

We have proposed an expressive binding specification language. In spite of its simplicity, it allows modeling many complex constructs. In particular, it supports constructs that bind a variable number of names, such as let and let rec. It also supports nonlinear patterns, that is, patterns where a single name can have multiple binding occurrences.

We have published [26] a tool and a library, collectively known as Cαml, which together allow turning binding specifications into executable Objective Caml code. Bundled with Cαml are a couple of toy demonstrations. One

implements a typechecker and evaluator for System $F_{\leq}$. It is inspired by the one written, using de Bruijn indices, by Pierce [21]. The other implements Hirschowitz *et al.*'s call-by-value reduction semantics for a calculus of mixin modules [10]. We deem both of these preliminary experiments to be successful: essentially no boilerplate code had to be manually written in order to deal with names.

No large scale application of Cαml exists yet, so it is too early to tell whether this approach is flexible and efficient enough to support realistic uses. We encourage potential users to have a try!

7.2 Limitations

There are specific activities that do not require "freshening" bound atoms upon opening abstractions. Weak reduction of closed λ-terms and typechecking are two examples. Then, automatic "freshening" is a waste of time. There are even cases where it gets in the way. For instance, many program analyses rely on the informal convention that "all bound names are fixed and pairwise distinct," which cannot be enforced in the presence of automatic "freshening." Yet, we insist that automatic "freshening" is a feature, not a limitation. Because it provides a vital safety guarantee, it intentionally cannot be disabled. If one wishes to avoid it, then one should work with slightly different terms, which one declares contain atoms, but no abstractions, so that all atoms are considered free and no "freshening" takes place. This should be no surprise: a "bound" name that is "fixed" is really a free name!

Because abstractions are translated to opaque structures, they cannot be taken apart via pattern matching. Instead, the programmer must explicitly call an *open* function. This is somewhat verbose and requires `match` constructs that involve deep patterns to be manually split into a cascade of shallow `match` constructs. This is an instance of a well-known tension between abstract types and pattern matching. Views [17] offer a potential solution to this issue. However, in the setting of an impure programming language, their semantics is difficult to design and to explain. They are currently not implemented in Objective Caml. In this respect, the Fresh Objective Caml approach is superior to that of Cαml: the compiler is modified to allow pattern matching against abstractions.

The functions that are used to open abstractions have impure semantics: they generate fresh atoms. Nothing guarantees that these atoms are used in a sensible way: one could, for instance, write a (meaningless) function that pretends to return the bound atoms of an expression, and get away with it. There clearly is a need for a type system that could prevent such misuse. Yet, the FreshML experience [24,32] suggests that this is a difficult problem. Perhaps it would be more easily addressed in the setting of a theorem prover, where it is fine to expect the user to explicitly state and prove properties, as opposed to that of a programming language in the ML tradition, where types

are supposed to be inferred. A typed calculus that attacks this problem was recently designed by Schürmann *et al.* [27].

7.3 From programming to proving

The program that we have carried out—designing a binding specification language and a tool that turns specifications into code—could be transposed into the world of theorem provers. What is needed is a binding specification language and a tool that turns specifications into theorems and proofs.

In fact, researchers in the theorem proving community have already identified this as a promising route. For instance, Pitts [23] wishes for "*an augmentation of the HOL or Isabelle datatype packages, allowing the user to declare a nominal signature and then have the principles of α-structural recursion and induction for that signature proved and ready to be applied.*" Similarly, Urban and Tasson [35] write: "*Ideally, a user just defines an inductive datatype and indicates where binders are—the rest of the infrastructure should be provided by the theorem prover.*" The authors of the PoplMark challenge [1] explain the need for automation.

In principle, it should be possible to write a tool that translates a version of our specification language (extended with features of the target language, such as dependent types) down to theorems and proofs. This appears to be an interesting direction for future research.

Acknowledgement

The author wishes to thank James Cheney, Jean-Christophe Filliâtre, Sébastien Hinderer, James Leifer, Randy Pollack, Yann Régis-Gianas, Didier Rémy, and Francesco Zappa-Nardelli, as well as the anonymous reviewers.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science. Springer Verlag, August 2005.

[2] Henk P. Barendregt. Functional programming and lambda calculus. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 321–363. Elsevier Science, 1990.

[3] Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.

[4] Robert Cartwright. Notes on object-oriented program design, January 2000.

[5] James Cheney. Scrap your nameplate. In *ACM International Conference on Functional Programming (ICFP)*, September 2005.

[6] James Cheney and Christian Urban. αProlog: A logic programming language with names, binding and α-equivalence. In *International Conference on Logic Programming (ICLP)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer Verlag, September 2004.

[7] Nicolaas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[8] Jacques Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, September 1998.

[9] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Haskell workshop*, 2000.

[10] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 64–78. Springer Verlag, April 2004.

[11] Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer Verlag, 2001.

[12] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, July 2004.

[13] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. Submitted, April 2005.

[14] Conor McBride and James McKinna. I am not a number: I am a free variable. In *Haskell workshop*, September 2004.

[15] Dale Miller. An extension to ML to handle bound variables in data structures. In *Logical Frameworks BRA Workshop*, May 1990.

[16] Gopalan Nadathur and Xiaochu Qi. Explicit substitutions in the reduction of lambda terms. In *International ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 195–206, August 2003.

[17] Chris Okasaki. Views for Standard ML. In *ACM Workshop on ML*, pages 14–23, September 1998.

[18] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *ACM Workshop on ML*, pages 77–86, September 1998.

[19] Emir Pašalić, Tim Sheard, and Walid Taha. DALI: An untyped, CBV functional language supporting first-order datatypes with binders (technical development). Technical Report 00-007, Oregon Graduate Institute, March 2000.

[20] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, June 1988.

[21] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[22] Andrew M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[23] Andrew M. Pitts. Alpha-structural recursion and induction. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science. Springer Verlag, August 2005.

[24] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *International Conference on Mathematics of Program Construction (MCP)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer Verlag, 2000.

[25] Gordon Plotkin. An illative theory of relations. In *Situation Theory and its Applications*, number 22 in CSLI Lecture Notes, pages 133–146. Stanford University, 1990.

[26] François Pottier. Cαml, June 2005.

[27] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The nabla-calculus. functional programming with higher-order encodings. Unpublished, September 2005.

[28] Tim Sheard. Accomplishments and research challenges in meta-programming. In *International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG)*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Verlag, 2001.

[29] Mark R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge, February 2005.

[30] Mark R. Shinwell. Fresh O'Caml: nominal abstract syntax for the masses. In *ACM Workshop on ML*, September 2005.

[31] Mark R. Shinwell and Andrew M. Pitts. Fresh Objective Caml user manual. Technical Report 621, University of Cambridge, February 2005.

[32] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *ACM International Conference on Functional Programming (ICFP)*, pages 263–274, August 2003.

[33] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, 1993.

[34] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[35] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science. Springer Verlag, 2005.

[36] Stephanie Weirich. A typechecker that produces a typed term from an untyped source. Part of the Glasgow Haskell compiler's test suite, September 2004.