Cαml Reference Manual
(version 20051129)

François Pottier
INRIA
Francois.Pottier@inria.fr

1. Foreword

Cαml (pronounced: "alphaCaml") is a tool that accepts a *binding specification* and turns it into Objective Caml type definitions and code. The generated code relies on a library known as *alphaLib*. Roughly speaking, a binding specification is a definition of one or several algebraic data types, enriched with information about *names* (henceforth referred to as *atoms*) and *binding*. This information gives rise to a notion of α-equivalence over the values that inhabit these types. The code produced by Cαml is intended to help deal with this notion in a safe and concise style.

This document is a reference manual. It is not a discussion of the problems raised by α-equivalence and of the various ways in which they can be addressed. Neither is it a tutorial introduction to Cαml. These topics are covered in a separate paper [1], which should be read first. Having a look at the demos that are shipped with Cαml is also recommended.

2. Terminology

Cαml values are split into *expressions* and *patterns*. Expressions are terms, that is, abstract syntax trees. Expressions contain *abstractions* inside which atoms can be bound. Inside abstractions are *patterns*. Patterns are also terms, but are slightly different. They cannot contain abstractions; that is, abstractions cannot be nested. Patterns can contain expressions, preceded with a *specifier* that tells whether the expression lies inside or outside the *scope* of the enclosing abstraction. The distinction between expressions and patterns is reflected in specifications, where each type is explicitly marked as an *expression type* or a *pattern type*.

Not all occurrences of an atom play the same role. For instance, consider the λ-term $a\,(\lambda a.a)$. The central occurrence of the atom a is meant to *bind* a in the body of the λ-abstraction: it is a *binding occurrence*. The first and last occurrences of a, on the other hand, are meant to *refer* to a previous binding occurrence of a: they are *referring occurrences*.

Both expressions and patterns can contain atoms, but they are interpreted differently. Occurrences of atoms that lie (directly) inside an expression are interpreted as referring occurrences, while occurrences of atoms that lie (directly) inside a pattern are interpreted as binding occurrences. In fact, this is a good way of summarizing the distinction between expressions and patterns.

It is common to need several distinct *sorts* of atoms—for instance, the abstract syntax of a typed programming language typically involves both term variables and type variables, which are separate. Cαml's specification language allows dealing with multiple sorts, as long as there is only a finite number of them.

3. Usage

By convention, binding specifications are stored in files whose name ends with `.mla`. Out of such a file, Cαml produces an Objective Caml compilation unit, that is, a pair of an `.ml` file and an `.mli` file. Both rely on the *alphaLib* library. Please have a look at the demos if you need help writing a Makefile in order to automate the compilation process. Read the generated `.mli` file—it is meant to be instructive. In order to

understand how atoms are implemented and what operations they support, consult the definition of the sub-module *AlphaLib.Signatures* (Appendix A).

4. Syntax of specifications

Notation Our terminal symbols are either literals, written in **bold** face, or one of *lid*, *uid*, and *qid*. The terminal symbol *lid* represents an Objective Caml identifier whose initial letter is lowercase, such as *beGentle*. The terminal symbol *uid* represents an Objective Caml identifier whose initial letter is uppercase, such as *Zero*. The terminal symbol *qid* represents an identifier whose initial letter is lowercase, possibly qualified with a module path, such as *List.map*. These three lexical categories are defined under the names *lowercase-ident*, *capitalized-ident*, and *value-path* in Objective Caml's manual (lexical conventions and names).

Non-terminal symbols are written in *italics*. The definition of a non-terminal symbol *nt* begins with *nt* ::= and goes on with a series of valid expansions for this symbol, each of which appears on a separate line. Each expansion is, to a first approximation, a sequence of terminal and non-terminal symbols. Square brackets $[\cdot]$ delimit an optional sub-sequence. Ellipses ... are used to indicate repetitions of a sub-sequence. Repetitions may or may not involve a delimiter. Although this notation is ambiguous, our syntax is simple enough that no difficulty should arise.

The syntax of specification (`.mla`) files appears in Figure 1. We now briefly explain each production.

Specification A specification consists of an optional prologue, followed by declarations. The order in which the declarations appear is irrelevant: all declarations are considered mutually recursive.

Prologue A prologue is a piece of Objective Caml text, delimited with square brackets. The prologue is copied verbatim to *both* of the generated files—that is, to the `.ml` file and to the `.mli` file—so it should make sense in both contexts. The prologue usually consists of **open** directives and of type definitions.

Declaration A declaration is a *sort declaration*, a *type declaration*, a *container declaration*, or an *identifier module declaration*.

◇ *Sort declarations.* A sort declaration introduces a new sort of atoms. It is possible to declare as many sorts as desired. Each sort declaration gives rise, in the generated code, to a distinct module. For instance, declaring "**sort** *termvar*" gives rise to a module named *Termvar*; declaring "**sort** *typevar*" gives rise to a module named *Typevar*. Both *Termvar* and *Typevar* have signature

AlphaLib.Signatures.Atom **with type** *identifier* = *Identifier.t*

The module type *Atom* is defined in *AlphaLib.Signatures*. The module *Identifier* is defined as part of the generated code; its identity can be controlled via an identifier module declaration. Note that *Termvar.Atom.t* and *Typevar.Atom.t* are distinct abstract types: that is, atoms of distinct sorts cannot be mixed.

◇ *Type declarations.* A type declaration introduces a new type. It is optionally parameterized by a sequence of Objective Caml type variables. These variables, if present, are allowed to appear inside the body of the declaration. There is, however, a restriction: all occurrences of a type *t* in the specification should carry the *same* sequence of parameters, that is, the same type variables, in the same order.

The **binds** clause is optional. If no clause is present, the type that is being declared is considered an expression type; otherwise, it is considered a pattern type. A **binds** clause mentions a set of sorts, which are considered bound by the pattern.

Each type declaration gives rise, in the generated code, to *two* type declarations, one of which lies at toplevel, and one of which lies inside the *Raw* sub-module. Thus, declaring a type *t* gives rise to two types named *t* and *Raw.t*.

◇ *Container declarations.* A container declaration introduces a new container. A container *t* is an Objective Caml type constructor with one parameter. It must represent a pure (that is, persistent) data structure with the semantics of a container: that is, values of type $\alpha\ t$ must represent collections of values of type α. It must come

specification ::=
[*prologue*] *declaration* . . . *declaration*

prologue ::=
[*arbitrary Objective Caml directives* **]**

declaration ::=
sort *sort*
type [*typevars*] *type* [**binds** *sort* **,** . . . **,** *sort*] **=** *body*
container *container* **with** *map* **and** *fold*
identifier module *module*

body ::=
branch . . . *branch*
factor ***** . . . ***** *factor*
{ *label* **:** *factor***;** . . . **;** *label* **:** *factor* }

branch ::=
| *data* [**of** *factor* ***** . . . ***** *factor*]

factor ::=
atom *sort*
[*arbitrary Objective Caml type* **]**
[*specifier*] [*typevars*] *type* [*container*]
< [*typevars*] *type* >
<**(** *type* **binds** *sort* **,** . . . **,** *sort***)** *body* >

specifier ::=
inner
outer
neutral

typevars ::=
typevar
(*typevar***,** . . . **,** *typevar* **)**

typevar ::=
' *lid*

sort, type, label ::=
lid

data, module ::=
uid

container, map, fold ::=
qid

Figure 1. Syntax of specifications

with *map* and *fold* functions, whose types must be

$$\begin{array}{rcl} \mathit{map} & : & \forall\alpha\beta.(\alpha \to \beta) \to \alpha\,t \to \beta\,t \\ \mathit{fold} & : & \forall\alpha\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\,t \to \alpha \end{array}$$

The names of these two functions must be supplied as part of the declaration. The containers *list* and *option* are predefined (and cannot be redefined).

◇ *Identifier module declarations.* An identifier module declaration specifies the name of an Objective Caml module, whose signature must be *AlphaLib.Signatures.Identifier*. This module is adopted as the definition of identifiers. It is provided as a parameter to the functor *AlphaLib.Atom.Make* in order to produce implementations of atoms. At most one identifier module declaration can appear in a specification. If none appears, then the default implementation *AlphaLib.Atom.String* is used. In this default implementation, identifiers are strings, and fresh identifiers are generated, when needed, by appending the decimal representation of an integer counter.

Body The body (that is, the right-hand side) of a type declaration can consist of a *sum type*, a *tuple type*, or a *record type*. A sum type consists of a list of branches. A tuple type consists of a list of factors. A record type consists of a list of factors, each of which carries a label.

Branch Each branch in a sum type consists of a data constructor, optionally followed by a list of factors.

Factor A factor is an *atom type*, a *foreign type*, a *type reference*, or an *abstraction type*. Not all factors are allowed in all contexts: some factors are valid only in the declaration of an expression type, while others are valid only in the declaration of a pattern type.

◇ *Atom types.* An atom type consists of the keyword **atom**, followed by a sort *sort*. It can appear both within expression types and within pattern types. Within an expression type, it is interpreted as a referring occurrence. Within a pattern type, it is interpreted as a binding occurrence; furthermore, in that case, the sort *sort* must be mentioned in the **binds** clause for that type.

◇ *Foreign types.* A foreign type is an arbitrary Objective Caml type expression, enclosed within square brackets. This expression can refer to any of the Objective Caml type variables currently in scope. For purposes of α-conversion, values of foreign type are ignored entirely: that is, they are considered not to contain any binding or referring occurrences of atoms. Values that contain modifiable state, or whose structure is not known, because it is represented by a type variable, are typically to be considered foreign. Foreign types are valid within expression and pattern types.

◇ *Type references.* A type reference primarily consists of (the name of) a type, which must be defined elsewhere in the specification. It is optionally preceded with a sequence of type variables, and optionally followed by a container. Type references can appear inside expression and pattern types. When within an expression type, no specifier must be given, and the reference must be again to an expression type. When within a pattern type, if a specifier is given, then the reference must be to an expression type: that is, specifiers can be thought of as end-of-abstraction marks. Otherwise, it must be again to a pattern type.

◇ *Abstraction types.* Abstraction types are valid within expression types only. There are two syntactic forms. In the simpler form, inside the angle brackets is (the name of) a type, which must be defined elsewhere in the specification and must be a pattern type. It is optionally preceded with a sequence of type variables.

We stress that the contents of an abstraction must be a type *name*: it cannot be, for instance, an anonymous product of factors. This is because a name is needed for the generated functions that allow creating and opening this abstraction.

The inconvenience of having to refer to a type name is somewhat relieved by the second, more elaborate form of abstraction types. In that form, a pattern type is introduced on the fly, and becomes the body of the abstraction. In other words, the definition of a pattern type is inlined into the abstraction. For instance, the abstraction type

$<$**(** *lambda* **binds** *var* **)** **atom** *var* ***** **inner** *term* $>$

is syntactic sugar for the abstraction type

$< lambda >$

together with the type declaration

type *lambda* **binds** *var* **=** **atom** *var* ***** **inner** *term*

Specifier A specifier precedes a reference to an expression type within a pattern type. In other words, a specifier marks the end of an abstraction. If the specifier is **inner**, then the expression is considered as lying *inside* the scope of the abstraction. If it is **outer**, then the expression is considered as lying *outside* the scope of the abstraction. When the expression contains, directly or indirectly, no atoms of the sorts bound by the abstraction, then it makes no difference whether it lies inside or outside the scope of the abstraction. In that case, and only in that case, the *neutral* specifier must be used. In other words, use of the *neutral* specifier is only permitted, and is required, in situations where **inner** and **outer** would have the same meaning.

5. Generated Code

The compilation unit produced by Cαml out of a specification file contains a number of Objective Caml module, type, value, and class definitions.

Modules For each atom sort declared in the specification, an "atom" module is generated. Each such module is produced by applying the functor *AlphaLib.Atom.Make* to the *Identifier* module. (The identity of the *Identifier* module can be controlled via an **identifier module** declaration.) As a result, every "atom" module has signature

AlphaLib.Signatures.Atom **with type** *identifier* = *Identifier.t*

Each "atom" module defines an abstract type of atoms and provides a number of operations involving atoms, sets and maps over atoms, maps of atoms to identifiers, and substitutions of atoms for atoms. The definition of the signature *AlphaLib.Signatures.Atom* can be found in Appendix A.

Types For each type *t* declared in the specification, *two* type definitions are produced. The *raw* version, named *Raw.t*, is concrete: atoms are identifiers. The *internal* version, named *t*, is abstract: atoms are abstract values of the appropriate "atom" module. The raw version is intended for use by parsers and pretty-printers; the internal form is intended for all other uses. Functions (*import_t* and *export_t*) are provided to convert back and forth between the two versions.

In the raw version, abstractions are transparent—that is, the angle brackets are erased. In the internal version, abstractions are opaque—that is, they are represented by abstract types. More precisely, in the internal version, for each pattern type *u* that appears within the body of an abstraction, *two* type definitions are produced: one, named *opaque_u*, is abstract, while the other, named *u*, is transparent. Functions (*create_u* and *open_u*) are provided to convert back and forth between the two forms.

Values For each expression type *t*, two functions *import_t* and *export_t* allow converting between raw and internal forms, that is, between *Raw.t* and *t*. The function *import_t* expects one or several mappings of identifiers to atoms—one mapping per relevant sort. Conversely, *export_t* expects one or several mappings of atoms to identifiers. These mappings are required when the terms to be converted contain free atoms or free identifiers. When converting closed terms, empty mappings can be provided.

For each type *t*, a function *subst_t* allows applying one or several substitutions of atoms for atoms to a term of type *t*—one substitution per relevant sort.

For each expression type *t*, a function *free_t* returns the sets of atoms that appear free in a term—again, one set per relevant sort. For each pattern type *u*, a function *bound_u* returns the sets of atoms that appear in a binding position in a term. A more complex function *bound_free_u* returns the sets of atoms that appear in a binding position, free in inner scope, or free in outer scope.

For each pattern type *u* that appears within the body of an abstraction, two functions *create_u* and *open_u* are produced, which convert back and forth between *u* and *opaque_u*. The function *create_u* has no runtime effect,

while *open_u* "freshens" all bound atoms: that is, it replaces them with fresh atoms. This enforces the informal convention that "bound names and free names must be chosen distinct".

Classes In order to help modularly define transformations and traversals over terms, Cαml produces two classes, named *map* and *fold*.

The class *map* contains one method for every type, data constructor, and record label in the specification. The method associated with type *t* has type $t \to t$. Its default implementation returns a copy of its argument. The copy is created via *self*-calls to the relevant methods for copying sub-terms, copying record fields and/or copying data constructor applications. The method associated with a data constructor *D* has the same type as *D*. Its default implementation applies *D* to a copy of its argument; again, the copy is obtained through appropriate *self*-calls. The method associated with a record field *f* of type τ has type $\tau \to \tau$. Its default implementation returns a copy of its argument, again created via appropriate *self*-calls.

If one were to create an object of class *map* via *new map*, each of its methods would behave as an identity function, whose argument is traversed and copied without change. The point is that it is now easy to create a sub-class of *map* where one or several methods are overridden. This results in application-specific behavior at certain nodes and default behavior at every other node. For instance, capture-free substitution of terms for variables can be defined in a few lines of code using this technique. This is illustrated in `demos/poplmark/core.ml`.

The class *fold* is very similar to *map*, except terms are only traversed, not copied. An accumulator is threaded through every call, that is, accepted and returned by every method. The class is parametric in the type of the accumulator.

6. Questions and Answers

◇ **How do I run experiments in the toplevel loop?** Create a specification file `foo.mla` and compile it, using Cαml and Objective Caml, so as to create `foo.cmi` and `foo.cmo`. Then, you can exploit the toplevel loop by entering the following directives:

```
#use "topfind";;
#require "alphaLib";;
#load "foo.cmo";;
```

If (through its prologue) your specification depends on other Objective Caml modules, you must load them before attempting to load `foo.cmo`.

If you want to check which operations are available over atoms, type

```
module V = Foo.Var;;
```

Assuming your specification contains the declaration "**sort** *var*", this causes the signature of the corresponding "atom" module to be displayed.

◇ **What about cyclic terms?** Abstract syntax trees are not meant to be cyclic. When using Cαml, the only way of creating cyclic terms is to exploit Objective Caml's liberal **let rec** construct. Don't do it.

◇ **What about sharing?** If your terms happen to have shared sub-terms, this is fine, but all sharing will be lost when the terms are copied—that is, when they are freshened, converted, renamed, etc.

◇ **What about marshaling?** Marshaling and unmarshaling of terms in internal form via *output_value* and *input_value* is fine, as long as the terms are *closed*, that is, have no free atoms.

◇ **How do I annotate nodes with mutable information?** The trick is to use an Objective Caml **ref** cell inside square brackets. It is even possible to abstract over the type of the desired information, by introducing a type parameter *'a*. A sample specification that uses this technique appears in Figure 2.

```
sort var

type 'a annotated_expression = {
    annotation: [ 'a ref ];
    body: 'a expression
  }

type 'a expression =
  | EVar of atom var
  | EAbs of <( lambda binds var ) atom var * inner 'a annotated_expression >
  | EApp of 'a annotated_expression * 'a annotated_expression
```

Figure 2. Annotating terms with mutable information

◇ **My lexer accepts identifiers that could end in terminating numerals. Wouldn't that interact badly with the *basename* and *combine* functions of the default *Identifier* module (Appendix A)?** No, this is fine. An identifier such as "*x3*" is turned by the *import* functions into an atom of basename "*x*". When this atom is later supplied to an *export* function, we obtain an identifier that begins with "*x*", followed by whatever integer is required to make this identifier unique. This could be just "*x*" or perhaps "*x24*".

In other words, dropping the "*3*"in "*x3*" when computing an atom's basename doesn't affect the fact that the atom is created unique. Furthermore, distinct atoms are mapped to distinct identifiers by the *export* functions. The basename mechanism is supposed to help these functions produce suggestive identifiers, but does not affect their soundness.

In fact, it would be possible to produce an implementation of the *Identifier* signature (Appendix A) where *basename* is a constant function, that is, where no basename information is recorded. In that case, the identifiers produced by the *export* functions would be isomorphic to integers. The composition of *import* and *export* would then act as a "name mangler". This could be useful when trying to obfuscate code!

A. Module *AlphaLib.Signatures*

This module defines a couple of signatures that specify the interaction between *alphaLib* and user programs.

A.1 Signature *AlphaLib.Signatures.Identifier*

This signature defines the operations that an implementation of identifiers must provide in order to appear in an **identifier module** declaration.

Identifiers are usually human-readable. In fact, the default implementation of identifiers, which is automatically supplied by Cαml when no **identifier module** declaration is made, equates identifiers with strings.

An implementation of atoms can be built on top of any implementation of identifiers via the internal functor *AlphaLib.Atom.Make*. This functor application is automatically performed by Cαml in order to produce each of the "atom" modules (one per sort).

In order to implement this signature, one must isolate a strict subset of identifiers, which we refer to as "base" identifiers. There must exist a function, referred to as *basename*, that maps arbitrary identifiers to base identifiers. This function does not have to (and usually cannot) be injective. (A function is injective when it maps distinct inputs to distinct outputs.) Nevertheless, the more information it preserves, the better. Conversely,, there must exist an injective function, referred to as *combine*, that maps a pair of a base identifier and an integer value to an identifier.

Internally, every atom records the image through *basename* of the identifier that it originally stood for. This base identifier plays no role in determining the identity of the atom: that is, it is not used when comparing two atoms. It is kept around for use when the atom is converted back to an identifier. At this point, it is combined, via *combine*, with a unique integer, in order to obtain suitably fresh identifier.

If *basename* and *combine* are properly chosen, then the final identifier that is printed "resembles" the one that was originally found. More precisely, it is desirable that the next two laws be satisfied:

$$\begin{aligned} \mathit{basename}\ (\mathit{combine\ identifier}\ i) &= \mathit{identifier} \\ \mathit{combine\ identifier}\ 0 &= \mathit{identifier} \end{aligned}$$

The first law states that the information added by combining an identifier with an integer *i* is exactly the information that is lost when applying *basename*. The second law states that combining the integer 0 with an identifier should have no effect. This is exploited to avoid needless renamings.

The default implementation of identifiers as strings defines base identifiers as strings that have no trailing digits. An identifier and an integer value are combined simply by appending a decimal representation of the latter to the former. In practice, this means that the identifier *x* will be successively renamed into *x*, *x1*, *x2*, *x3*, and so on.

module type *Identifier* = **sig**

t is the type of identifiers.

type *t*

Identifiers must be comparable. As usual in Objective Caml, *compare id1 id2* must return a negative integer if *id1* is less than *id2*, a positive integer if *id2* is less than *id1*, and zero otherwise.

val *compare* : *t* → *t* → *int*

basename and *combine* are described above.

val *basename* : *t* → *t*
val *combine* : *t* → *int* → *t*

The sub-module *Map* provides maps whose keys are identifiers. It is usually produced by applying the standard library functor *Map.Make* to the type *t* and the function *compare* above. This sub-module is used by the functions that convert back and forth between raw and internal forms, that is, between atoms and identifiers.

module *Map* : *Map.S* **with type** *key* = *t*

end

A.2 Signature *AlphaLib.Signatures.Atom*

This signature specifies the operations that every "atom" module provides. These operations are grouped into several sub-modules that provide:

- basic operations on atoms;
- sets of atoms;
- maps of atoms to arbitrary data;
- maps of atoms to identifiers;
- substitutions of atoms for atoms.

---

module type *Atom* = **sig**

The type *identifier* is the type of the identifiers on top of which this implementation of atoms is built.

type *identifier*

The sub-module *Atom* offers an abstract type of atoms. Atoms are abstract entities that support two (classes of) operations, namely creation of fresh atoms and comparison of two atoms.

Every atom carries a unique integer, which can be viewed as its identity. This integer is used in comparisons. A global integer counter is maintained and incremented when fresh atoms are created.

Every atom also carries a base name, that is, an identifier. This identifier is not in general unique, and is not part of the atom's identity. It is used when converting atoms back to identifiers: see, for instance, *AtomIdMap.add*.

When a fresh atom is created, its base name is taken either from an existing identifier or from an existing atom. Two functions, *freshb* and *fresha*, are provided for this purpose.

module *Atom* : **sig**

t is the type of atoms.

type *t*

The call *freshb identifier* produces a fresh atom whose base name is that of *identifier*.

val *freshb* : *identifier* → *t*

The call *fresha a* produces a fresh atom whose base name is that of *a*.

val *fresha* : *t* → *t*

Atoms can be tested for equality, for ordering, and hashed. The user is warned against careless use of *compare* and *hash*. Atoms are renamed during α-conversion, which affects their relative ordering and their hash code. It is fine to use these operations as long as one guarantees that no renaming takes place.

val *equal* : *t* → *t* → *bool*
val *compare* : *t* → *t* → *int*
val *hash* : *t* → *int*

It is possible to retrieve an atom's identity and base name. There is in general no good reason of doing so, except for debugging purposes.

val *identity* : $t \rightarrow int$
val *basename* : $t \rightarrow identifier$

The exception *Unknown* is raised by *AtomIdMap.lookup*.

exception *Unknown* **of** *t*

end

The sub-module *AtomSet* offers a representation of finite sets of atoms.

module *AtomSet* : **sig**

t is the type of sets of atoms.

type *t*

empty is the empty set.

val *empty* : *t*

singleton a is the singleton set $\{a\}$.

val *singleton* : *Atom.t* $\rightarrow$ *t*

add x s is $(\{x\} \cup s)$.

val *add* : *Atom.t* $\rightarrow$ *t* $\rightarrow$ *t*

union s1 s2 is $(s1 \cup s2)$.

val *union* : *t* $\rightarrow$ *t* $\rightarrow$ *t*

inter s1 s2 is $(s1 \cap s2)$.

val *inter* : *t* $\rightarrow$ *t* $\rightarrow$ *t*

diff s1 s2 is $(s1 \setminus s2)$.

val *diff* : *t* $\rightarrow$ *t* $\rightarrow$ *t*

mem a s is **true** if and only if *a* is a member of *s*.

val *mem* : *Atom.t* $\rightarrow$ *t* $\rightarrow$ *bool*

is_empty s is **true** if and only if *s* is the empty set.

val *is_empty* : *t* $\rightarrow$ *bool*

disjoint s1 s2 is **true** if and only if the sets *s1* and *s2* are disjoint, that is, if and only if their intersection is empty.

val *disjoint* : *t* $\rightarrow$ *t* $\rightarrow$ *bool*

equal s1 s2 is **true** if and only if *s1* and *s2* are extensionally equal, that is, if and only if they have the same members.

val *equal* : *t* $\rightarrow$ *t* $\rightarrow$ *bool*

subset s1 s2 is **true** if and only if *s1* is a subset of *s2*, that is, if and only if every member of *s1* is also a member of *s2*.

val *subset* : $(t \rightarrow t \rightarrow bool)$

The call *iter f s* has the effect of applying the function *f* in turn to every member of *s*.

val *iter* : $(Atom.t \rightarrow unit) \rightarrow t \rightarrow unit$

The call *fold f s accu* has the effect of applying the function *f* in turn to every member of *s* and to an accumulator whose value, threaded through the calls, is initially *accu*. Its result is the final value of the accumulator.

val *fold* : $(Atom.t \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

end

The sub-module *AtomMap* offers a representation of finite maps whose keys are atoms and whose data can have arbitrary type α.

module *AtomMap* : **sig**

α *t* is the type of maps of atoms to data of type α.

type α *t*

empty is the empty map.

val *empty* : α *t*

singleton a d is the singleton map that maps atom *a* to datum *d*.

val *singleton* : $Atom.t \rightarrow \alpha \rightarrow \alpha$ *t*

add a x m is the map that maps atom *a* to datum *d* and elsewhere behaves like *m*.

val *add* : $Atom.t \rightarrow \alpha \rightarrow \alpha$ *t* $\rightarrow \alpha$ *t*

union m1 m2 is the map that behaves like *m2* where *m2* is defined and elsewhere behaves like *m1*. In other words, the bindings in *m2* take precedence over those in *m1*.

val *union* : α *t* $\rightarrow \alpha$ *t* $\rightarrow \alpha$ *t*

lookup a m returns the datum associated with atom *a* in the map *m*, if defined, and raises the exception *Not_found* otherwise.

val *lookup* : $Atom.t \rightarrow \alpha$ *t* $\rightarrow \alpha$

is_empty m is **true** if and only if *m* is the empty map.

val *is_empty* : α *t* $\rightarrow$ *bool*

map f m is the map obtained by composing the function *f* with the map *m*, that is, the map that maps an atom *a* to $(f\ d)$ when *m* maps *a* to *d*.

val *map* : $(\alpha \rightarrow \beta) \rightarrow \alpha$ *t* $\rightarrow \beta$ *t*

end

The sub-module *AtomIdMap* offers finite maps of atoms to identifiers, with the property that every atom is mapped to a distinct identifier. This invariant is enforced by having the library pick a unique identifier when a new atom is added to the domain of the map. That is, the client does not control which identifiers are picked.

module *AtomIdMap* : **sig**

t is the type of maps.

type *t*

empty is the empty map.

val *empty* : *t*

add a m is a map that maps the atom *a* to a unique identifier (that is, an identifier not in the codomain of *m*) and elsewhere behaves like *m*. The base name of *a* is used when picking this identifier. The atom *a* must not be a member of the domain of *m*.

val *add* : *Atom.t* → *t* → *t*

add_set s m is the map obtained by successively *add*ing every member of the atom set *s* to the map *m*.

val *add_set* : *AtomSet.t* → *t* → *t*

lookup a m returns the identifier associated with the atom *a* in the map *m*, if defined, and raises the exception *Atom.Unknown* otherwise.

val *lookup* : *Atom.t* → *t* → *identifier*

end

The sub-module *Subst* offers substitutions of atoms for atoms. These are total mappings of atoms to atoms that behave as the identity outside of a finite set of atoms, known as their domain.

module *Subst* : **sig**

t is the type of substitutions.

type *t*

id is the identity substitution.

val *id* : *t*

is_id subst is **true** if and only if *subst* is the identity substitution.

val *is_id* : *t* → *bool*

singleton a b is the singleton substitution that maps atom *a* to atom *b*.

val *singleton* : *Atom.t* → *Atom.t* → *t*

add a b subst is the substitution that maps atom *a* to atom *b* and elsewhere behaves like *subst*.

val *add* : *Atom.t* → *Atom.t* → *t* → *t*

union subst1 subst2 is the substitution that behaves like *subst2* on its domain and elsewhere behaves like *subst1*. In other words, the bindings in *subst2* take precedence over those in *subst1*.

val *union* : *t* → *t* → *t*

compose subst1 subst2 is the composition of *subst1* with *subst2*, that is, the substitution that maps every atom *a* to *subst1* (*subst2* (*a*)).

val *compose* : *t* → *t* → *t*

freshen s subst is a substitution that maps every atom *a* in the set *s* to a fresh atom (obtained via *Atom.fresha a*) and elsewhere behaves like *subst*.

val *freshen* : *AtomSet.t* → *t* → *t*

lookup a subst is the image of *a* through *subst*. It is never undefined, since substitutions are viewed as total mappings.

val *lookup* : *Atom.t* → *t* → *Atom.t*

end

end

References

[1] François Pottier. An overview of Cαml. Submitted, June 2005.