UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports Techniques

N°??

Programme 2

Calcul symbolique, Programmation et Génie logiciel

AN IMPLEMENTATION OF CAML-LIGHT WITH EXISTENTIAL TYPES

Michel Mauny
François Pottier

Octobre 1993

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

An implementation of Caml-Light with existential types

Abstract

This work is based on a proposal by Läufer and Odersky [3]. They show that it is possible to add existential types to an ML-like language without even modifying its syntax. ML's strong typing properties are of course retained. We implemented their proposal into Caml-Light 0.6 [4], thus making it possible to write realistic programs using existential types.

The first part of this paper describes this simple form of existential types and showss how to use them in our enhanced version of Caml-Light. We then give several examples demonstrating their usefulness. Finally, a more technical section gives an overview of the changes made to the compiler and discusses some technical issues.

Une implémentation de Caml-Light avec types existentiels

Résumé

Ce travail est fondé sur une proposition de Läufer et Odersky [3] montrant l'adjonction de types existentiels aux langages ML sans modification de la syntaxe de ces langages. Les propriétés du typage fort de ML sont bien sûr préservées. Nous avons implémenté cette proposition en Caml-Light 0.6, permettant ainsi l'écriture de programmes de taille réelle utilisant ces types existentiels.

La premiere partie de ce document décrit cette forme simple de types existentiels, et montre leur utilisation dans le langage ainsi étendu. Après quoi nous donnons quelques exemples démontrant l'utilité de ces types existentiels. Enfin, une section plus technique décrit bièvement les principaux changements que nous avons apportés au compilateur et aborde quelques points techniques.

Authors addresses: Michel Mauny, INRIA-Rocquencourt, projet Cristal, `Michel.Mauny@inria.fr` and François Pottier, Ecole Normale Supérieure, Paris, `Francois.Pottier@ens.fr`

Introduction

This work is based on a proposal by Läufer and Odersky [3]. They show that it is possible to add existential types to an ML-like language without even modifying its syntax. ML's strong typing properties are of course retained. We implemented their proposal into Caml-Light 0.6 [4], thus making it possible to write realistic programs using existential types.

Existential types, even under such a simple form provide an abstraction mechanism that is already useful for programming libraries (such as the `set` library given in section 2.3). We also believe that our implementation provides a nice platform for conducting small-sized experiments with the abstraction mechanism provided, and is also an interesting tool for teaching important concepts such as abstract data types in functional programming courses.

The first part of this paper describes this simple form of existential types and show how to use them in our enhanced version of Caml-Light. We then give several examples demonstrating their usefulness. Finally, a more technical section gives an overview of the changes made to the compiler and discusses some technical issues.

1 Using existential types in Caml-Light

Type specifications for variables of a universally quantified type have the following form, for any type expression $t(a)$:

$$\rho : \forall a.t(a) \qquad\qquad (e.g.\ id : \forall a.a \rightarrow a)$$

By analogy with universal quantification, we give a meaning to existentially quantified types. By stating that a value ρ has the existential type $\exists\alpha.t(\alpha)$, we mean that for some unknown type $\hat{\alpha}$, ρ has type $t(\hat{\alpha})$.

Läufer and Odersky proposed an interpretation of type variables occurring free in ML type definitions [2, 3]: such variables should be thought of as being existentially quantified, and they show that, with such an interpretation, we obtain a simple but still useful form of abstract data types. We explain below our adaptation and implementation of their work in the Caml-Light context.

1.1 Type declarations

In Caml-Light, a type declaration is of the form

`type` [*argument-list*] *identifier* `=` *body* `;;`

where *argument-list* is the list of formal type parameters, which may appear free in the type expression *body*. The resulting type is universally quantified over these parameters, and no other type variables may appear free in *body*.

Without altering the syntax of the type declaration, we now give a meaning to type variables that appear free in *body*, but not in the type parameter list. Following Läufer and Odersky, we interpret such type variables as existentially quantified.
For instance,

```
#type key = Key of 'a * ('a -> int);;
Type key defined.
```

describes a type with one value constructor, `Key`, whose arguments are pairs of a value of type `'a` and a function of type `'a -> int`. Here we don't know anything about `'a`, except that the value is of the same type `'a` as the function domain.

1.2 Constructing values with existential types

Values of existential types are constructed using usual Caml-Light syntax. For instance, the expressions

```
#Key (3, fun x -> x + 3);;
- : key = Key (<abstract>, <fun>)

#Key ([0; 1], list_length);;
- : key = Key (<abstract>, <fun>)
```

both have type `key`. Note that no argument types appear in the result types of these expressions: the underlying types (respectively `int` and `int list`) have been "forgotten". Caml only displays the `<abstract>` keyword: since those values are of unknown types, they cannot be displayed.

On the other hand, this is incorrect:

```
#Key (true, list_length);;
> Toplevel input:
>Key (true, list_length);;
>           ^^^^^^^^^^^
> Expression of type 'a list -> int
> cannot be used with type bool -> int
```

since the type of `true` and the domain type of `list_length` don't match.

1.3 Decomposing values with existential types

1.3.1 Pattern-matching on values with existential types

Besides constructing values with existential types, we can decompose them using a `match` (or `let`, or `fun`) construct, such as:

```
#let x = Key ("abc", string_length);;
x : key = Key (<abstract>, <fun>)

#match x with Key (v, f) -> f v;;
- : int = 3
```

What happens when a value with an existentially quantified type is decomposed? Suppose we write "`match` e `with` p_1 `->` e_1 `|` $\ldots p_n$ `->` e_n ", where e has type $\exists\alpha_1 \ldots \exists\alpha_n . t(\alpha_1 \ldots \alpha_n)$. When Caml's typechecker encounters this expression, for each pattern p_i, it takes an instance of this type, i.e. it creates n abstract types $\hat{\alpha}_1^i, \ldots, \hat{\alpha}_n^i$ (called Skolem type constructors) and assigns type $t(\hat{\alpha}_1^i, \ldots, \hat{\alpha}_n^i)$ to p_i in order to conduct the pattern matching. A Skolem constructor $\hat{\alpha}$ has the property that it can match no types but itself. This reflects the fact that a Skolem type constructor represents an unknown type; hence it cannot be unified with other types.

This process is illustrated in the following pattern-matching `let` construct:

```
#let (Key (a, b)) = x;;
b : ?A -> int = <fun>
a : ?A = <abstract>
```

Here `x` has type `key`, which is the same as $\exists \alpha. key(\alpha * (\alpha \rightarrow int))$. Thus the typechecker creates a Skolem constructor `?A`, `a` is given type `?A` and `b` is given type `?A -> int`. The Caml-Light compiler uses `?A`, `?B`, ..., (upper-case letters) for printing toplevel Skolem constructors, and `?a`, `?b`, ..., (lower-case letters) for local ones (cf. error messages below).

On the other hand, this is incorrect:

```
#let x = Key (3, succ) in
#       match x with Key (v, _) -> v + 6;;
> Toplevel input:
>       match x with Key (v, _) -> v + 6;;
>                                  ^
> Expression of type ?a
> cannot be used with type int
```

The `match` construct gives `v` a unique, abstract type `?a`, which can match no types but itself. When the typechecker encounters `v + 6`, it tries to unify `?a` with `int`, which is illegal. Hence evaluating `v + 6` is impossible, even though `v` has a concrete value (3) of type `int`.

1.3.2 Scope restrictions

In order for the type system to be safe, the scope of existential type variables must be restricted. Läufer and Odersky suggest to put the scope limit to the end of pattern-matching constructs (In Caml, `let`, `match`, `fun` etc.). Our scope limitation is only on `fun` constructs: using our previous notations, this means that `?a` must not appear in the result type of all enclosing `fun` constructs. Without this rule, it is easy to produce well-typed programs that lead to crashes (see section 3.1).

Here are some examples of prohibited expressions. This one

```
#fun x -> match x with Key (v, f) -> f;;
> Toplevel input:
>fun x -> match x with Key (v, f) -> f;;
>                                    ^
> Expression has an existential type escaping out of its scope.
```

is invalid because the result type of the `fun` construct would be `key -> ?a -> int`.
So is this one:

```
#let g h = fun (Key (v, f)) -> h v;;
> Toplevel input:
>let g h = fun (Key (v, f)) -> h v;;
>                                ^
> Expression has an existential type escaping out of its scope.
```

Even though the abstract type `?a` doesn't appear in the type of `fun Key (v, f) -> h v`, allowing such an expression would give type `?a -> 'a` to `h`, thus letting `?a` escape out of its scope.

Existential type variables can appear in the types of variables that have been bound at toplevel, such as `v` in

```
#let (Key (v, f)) = x;;
f : ?B -> int = <fun>
v : ?B = <abstract>
#let v' = let (Key(v,_)) = x in v;;
v' : ?C = <abstract>
```

since the scope of such variables is the whole toplevel session (or the current module). In order to ensure the safety of the type system, values with existential types cannot be exported out of a module (cf. section1.4).

1.3.3 Existential types and the dot notation

In regular Caml-Light, the dot notation `expr.Field` is totally equivalent to the pattern-matching construct:

```
let {Field = f} = expr in f
```

For example, given the following type and value definition

```
#type obj = {Data : 'a; Op : 'a -> int};;
Type obj defined.
#let o = {Data="abcd"; Op=string_length};;
o : obj = {Data=<abstract>; Op=<fun>}
```

we can access the `Data` and `Op` fields with the dot notation:

```
#let d1 = o.Data
#and d2 = o.Data
#and op = o.Op;;
d1 : ?D = <abstract>
d2 : ?E = <abstract>
op : ?F -> int = <fun>
```

Unfortunately, since we opened three times the `o` value, values `d1`, `d2` and `op` cannot communicate:

```
#op d1;;
> Toplevel input:
>op d1;;
>   ^^
> Expression of type ?D
> cannot be used with type ?F
#[d1;d2];;
> Toplevel input:
>[d1;d2];;
>    ^^
> Expression of type ?E
> cannot be used with type ?D
```

This example shows that the dot notation is very impractical (if not useless) in presence of existential types, since each access to a given field of the same value receives a type which is incompatible with those of all other accesses.

Solutions to this problem have been proposed [1] by considering an *access path equivalence* that identifies several accesses to the same variable as belonging to a single pattern-matching construct. The trick is to virtually decompose that variable only once, at its binding occurence. Given such an equivalence, it would be possible to assign the same existential type to equivalent accesses.

At present we have not tried to tackle this problem. In our implementation it is necessary to explicitly use a matching construct large enough in order to open a package. This matching may be done at top-level, which makes things slightly easier.

1.4 Existential types, modules and separate compilation

If we wanted to be able to export values with existential types out of a module, we would need a kind of *stamp relocalization* in order to avoid confusing existential types coming from different modules. Instead of implementing such a mechanism, we chose to avoid exporting such values. Since there is no syntactic support for interface specifications such as

```
value r : ?A
```

we had only two cases to consider:

- module `mod.ml` is being compiled, but does not possess an interface `mod.mli`;
- and `mod.ml` has an interface `mod.mli`, but is being compiled with `-g` option.

These two situations are similar: in both cases a default interface is generated with *all* values and types being exported. If an existential type occurs in the type of one of the exported values, a self-explanatory warning is generated by the compiler, as shown by the following Unix session:

```
% cat mod.ml
type key = Key of 'a * ('a -> int);;
let (r,f) = let (Key(r,f)) = Key(1, succ) in (r,f);;
% camlc -c -g mod.ml
# Warning: value f not exported (its type contains existential variables)
# Warning: value r not exported (its type contains existential variables)
```

The module `mod.ml` has been successfully compiled, generating files `mod.zo` (object code) and `mod.zix` (extended compiled interface). However, values `r` and `f` are inaccessible.

2 Applications – Code examples

2.1 Existential types as abstract types

What new opportunities are provided by existential types? Not all existential types turn out to be useful. For example, if we define

```
#type dumb = K of 'a;;
Type dumb defined.
```

and if we have an object of type `dumb`, we have absolutely no way of manipulating it (except passing it around) because we have no information about it.

Existentially typed objects can however be useful if they are sufficiently structured. For example, let us define

```
#type obj = {Data : 'a; Op : 'a -> int};;
Type obj defined.
```

An object of type `obj` provides sufficient structure to allow us to compute with it. We can define

```
#let f {Data = d; Op = o} = o d;;
f : obj -> int = <fun>
```

`f` is a function of type `obj -> int` and can be applied to any value of type `obj`, independently of its actual concrete representation.

In fact, `obj` is a simple example of an *abstract type* packaged with its set of operations. The abstract type is represented by the `'a` variable in the type declaration; here the type is provided with two operators, `data` of type `'a` (actually a constant) and `op` of type `'a -> int`.

When given a value – one could say a package – of type `obj`, the user will not be able to take advantage of the way it is implemented, because its actual type is hidden. Thus, existential types provide an easy way to achieve type abstraction.

2.2 A sample abstract type: `stack`

We give here an implementation of stacks in Caml-Light, which serves as a very simple example of type abstraction using existential quantification.

```
#(* Type and exception definitions *)
#type 'a stack = {
#  mutable Repr : 'b;
#          Push : 'a -> 'b -> 'b;
#          Pop : 'b -> 'b * 'a;
#          Size : 'b -> int
#};;
Type stack defined.

#exception Empty;;
Exception Empty defined.

#exception Full;;
Exception Full defined.

#(* Generic functions *)
#let gpush elem = fun {Repr = r; Push = push_func} ->
#  r <- push_func elem r;;
gpush : 'a -> 'a stack -> unit = <fun>

#let gpop = fun {Repr = r; Pop = pop_func} ->
#  let (r', elem) = pop_func r
#  in r <- r';
```

```
#     elem;;
gpop : 'a stack -> 'a = <fun>

#let gsize = fun {Repr = r; Size = size_func} ->
#  size_func r;;
gsize : 'a stack -> int = <fun>

#(* A list-based implementation *)
#let lpush elem lst = elem::lst;;
lpush : 'a -> 'a list -> 'a list = <fun>

#let lpop lst = match lst with
#  [] -> raise Empty
#| elem::rest -> (rest, elem);;
lpop : 'a list -> 'a list * 'a = <fun>

#let stack1 = ({
#        Repr = [];
#        Push = lpush;
#        Pop = lpop;
#        Size = list_length} : int stack)
#;;
stack1 : int stack = {Repr=<abstract>; Push=<fun>; Pop=<fun>; Size=<fun>}

#(* An array-based implementation *)
#let max = 1000;;
max : int = 1000

#let apush elem (index, array) =
#  if index = max then raise Full
#  else vect_assign array (index+1) elem;
#       (index+1, array);;
apush : 'a -> int * 'a vect -> int * 'a vect = <fun>

#let apop (index, array) =
#  if index = 0 then raise Empty
#  else ((index-1, array), vect_item array index);;
apop : int * 'a vect -> (int * 'a vect) * 'a = <fun>

#let stack2 = {Repr = (0, make_vect max 0);
#         Push = apush;
#         Pop = apop;
#         Size = fst}
#;;
stack2 : int stack = {Repr=<abstract>; Push=<fun>; Pop=<fun>; Size=<fun>}
```

This example shows two advantages of existential quantification.

First, a value of type `stack` is opaque, i.e. the user doesn't have direct access to its components. Since he doesn't have any knowledge of the stack's internal structure, he is forced to use the stack through the mechanisms we provided, i.e. the `Push`, `Pop` and `Size` fields.

Second, although `stack1` and `stack2` have totally different implementations, they both are of type `int stack` and they look identical to the user (except that they have different overflow conditions). The user can use them exactly the same way, using the generic functions `gpush`, `gpop` and `gsize`. It is even possible to build and use heterogeneous structures:

```
#let het_list = [stack1; stack2]
#in do_list (gpush 3) het_list;;
- : unit = ()
```

The `gpush` function accepts any arguments of type `key`, regardless of their concrete representation, and performs what could be called a *dynamic dispatching* based on its argument's type. Dynamic dispatching is an important feature of object-oriented languages. Hence existential types allow for a new kind of polymorphism, which resembles that found in object-oriented languages. Of course we are still far away from the power of these languages, since no inheritance mechanism is present.

2.3 An interesting case: the `set` library

In mid-93, there was a discussion in the Caml mailing list about the new `set` module, to be introduced in Caml-Light release 0.6. This module provides basic operations on sets.

In order to achieve good efficiency, sets are represented internally as balanced binary trees. But balanced trees can be built only over an ordered set of elements. Hence, in order to create a set, an ordering function must be specified.

Some people proposed using a generic ordering primitive, which would compare two elements by looking at their physical structure. This primitive can be easily programmed, in the same way as the generic equality primitive. The problem is that, in most cases, this primitive is inadequate. For instance, if we create a set of stamped objects, generic ordering will insist on comparing all fields of the object, possibly looping, while all is needed is to compare the stamp fields. It doesn't work with sets of sets, either: since sets have a complex structure, two sets can be equal and yet have a different physical structure.

Hence, it appears clear that the ordering function must be chosen by the user. It has been proposed to give the following interface to the module:

```
type 'a t;; (* The type of sets containing elements of type 'a *)
type 'a ordering == 'a -> 'a -> int;;

value empty: 'a t
  and mem: 'a ordering -> 'a -> 'a t -> bool
  and add: 'a ordering -> 'a -> 'a t -> 'a t
  and union: 'a ordering -> 'a t -> 'a t -> 'a t
;;
```

There were other operations, but they are omitted here for the sake of brevity.

An ordering function is a two-argument function `f` such that `f e1 e2` is zero if `e1` and `e2` are equal, negative if `e1` is smaller than `e2`, and positive otherwise. In this scheme, the user is responsible for always using the same function to operate on a given set. Otherwise, the above functions could produce incorrect results.

This works, but it is not foolproof at all. Even the most experienced programmer could pass the wrong ordering function by mistake and confuse the library functions. Besides, always passing an ordering function as first argument is cumbersome.

Therefore, it was suggested to write the interface this way:

```
type 'a t;;
type 'a ordering == 'a -> 'a -> int;;

value empty: 'a ordering -> 'a t
  and mem: 'a -> 'a t -> bool
  and add: 'a -> 'a t -> 'a t
  and union: 'a t -> 'a t -> 'a t
;;
```

and the implementation of type `'a t` would then look like:

```
type 'a t = {Order: 'a ordering; Data: (* whatever *) };;
```

In this scheme, the ordering function is specified only once – when creating the set with the `empty` function. It is then stored inside the set structure. This seems to solve the problems we previously encountered. First, it suppresses the need to pass an ordering function every time. Second, since the function is stored inside each set, it is impossible to specify the wrong ordering.

But there remains a slight problem with functions that take two set arguments, such as `union`. These functions work correctly only if their two set arguments have the same ordering function – and yet nothing prevents the user from passing incompatible sets. Hence, this interface is still not foolproof.

Existential types provide a way to solve this problem. The following example demonstrates it. Only the implementation is shown, along with the toplevel's output. If we were to write an interface, only `set` and `buildset` would appear in it.

```
#type 'a set = {
#           Empty : 'b;
#           Mem : 'a -> 'b -> bool;
#           Add : 'a -> 'b -> 'b;
#           Union : 'b -> 'b -> 'b
#        }
#;;
Type set defined.

#let list_member compare x l =
#  let rec rec_member = function
#    [] -> false
#  | elem::rest -> (match compare x elem
#                   with 0 -> true
#                     | _ -> rec_member rest)
#  in rec_member l
#;;
list_member : ('a -> 'b -> int) -> 'a -> 'b list -> bool = <fun>
```

```
#let list_union compare set1 =
#  let rec rec_union accu = function
#     [] -> accu
#  | elem::rest -> rec_union (if list_member compare elem accu then accu
#                             else elem::accu) rest
# in rec_union set1
#;;
list_union : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list = <fun>
```

```
#let buildset compare =
#  {Empty = [];
#   Mem = list_member compare;
#   Add = (fun x l -> if list_member compare x l then l else x::l);
#   Union = list_union compare}
#;;
buildset : ('a -> 'a -> int) -> 'a set = <fun>
```

In order to operate on sets, it is necessary to create an object of type `set`, which provides all basic set operations. There is only one way to create such a package, namely through the `buildset` function. When using this function it is necessary to specify an ordering function, `compare`.

We have now solved the problem of functions with two set arguments. Suppose we create a package and open it with a decomposing `match` or `let`. A unique Skolem constructor, which we may call `?a`, is created. The `union` function will then only accept arguments that have type `?a`. Such arguments must have been created with the same package – otherwise their type would be *another* Skolem constructor. This is demonstrated below:

```
#let package = buildset (fun i j -> (i - j) mod 7);;
package : int set = {Empty=<abstract>; Mem=<fun>; Add=<fun>; Union=<fun>}
```

```
#let {Empty = e; Mem = m; Add = a; Union = u} = package;;
u : ?G -> ?G -> ?G = <fun>
a : int -> ?G -> ?G = <fun>
m : int -> ?G -> bool = <fun>
e : ?G = <abstract>
```

```
#m 2 (u (a 1 e) (a 9 e));;
- : bool = true
```

Now let's create another package and try to use them together:

```
#let {Empty = e2; Mem = m2; Add = a2; Union = u2} =
#  buildset (fun i j -> i - j);;
u2 : ?H -> ?H -> ?H = <fun>
a2 : int -> ?H -> ?H = <fun>
m2 : int -> ?H -> bool = <fun>
e2 : ?H = <abstract>
```

```
#u (a 1 e) (a2 1 e2);;
> Toplevel input:
```

```
>u (a 1 e) (a2 1 e2);;
>             ^^
> Expression of type int -> ?H -> ?H
> cannot be used with type int -> ?H -> ?G
```

As one can see, the typechecker tries to unify two different Skolem constructors and fails. Note that opening a package is rather cumbersome, but it is done only once at toplevel.

As a conclusion, this is an interesting application of existential type variables. Not only can they be used as abstract types – in some cases they bring a real improvement over Caml-Light's plain typing system.

3 Implementation

This section discusses the changes made to the compiler, as well as some implementation choices.

3.1 Scope restrictions

Skolem type constructors have a scope, and they cannot be exported out of this scope. We give here a code sample that illustrates the need to restrict the scope of Skolem type constructors. Then we discuss how we enforced this restriction in the implementation.

Reproduced below is the output of a hypothetical, non-restrictive version of Caml-Light.

```
#let x = Key (3, succ)
#and y = Key (true, fun x -> 1);;
x : key = Key (<abstract>, <fun>)
y : key = Key (<abstract>, <fun>)

#let decompose (Key (a, b)) = (a, b);;
decompose : key -> ?A * (?A -> int) = <fun>

#let (rx, fx) = decompose x
#and (ry, fy) = decompose y
#in fx ry;;
Bus error
```

The problem arises from the fact that when we define the `decompose` function, it is typed statically, i.e. every time we apply `decompose` to a value of type `key`, the result is considered as being of type `?A * (?A -> int)`, where `?A` remains the same Skolem constructor every time. This allows us to confuse the typechecker into thinking that `fx` can be applied to `ry`, thus causing a critical error. This is of course incorrect: a new Skolem constructor should be created for each application of the function; but this requires an explicit existential quantification of Skolem constructors. In other words, the type of `decompose` should be `key ->` $\exists\alpha.\alpha$`*(`α`->`int`)`. Since there is no support for such a quantification, it is necessary to make sure that Skolem type constructors remain local to the `fun` construct that created them.

Although the scope restrictions given in Läufer and Odersky's original work concerned all pattern-matching constructs (`let` and `match`), we found that the only necessary restriction concerned `fun` constructs. As shown by the example above, problems occur if we are able to match *several* different values while producing the *same* Skolem constructors each time. This is the typical behaviour of the `fun` construct. Other matching constructs (`let` and `match`[1]) apply to only one textual expression: the scope of Skolem constructors produced by such constructs can therefore be extended to the enclosing `fun`, if any.

We found a very simple and effective way of enforcing this restriction. In the original version of Caml-Light, every type variable carries a number, called its *binding level*. It is the nesting level of the `let` construct which caused this variable to appear, and it is used in order to correctly generalize type variables after typing a `let ... in` clause. We chose to make these numbers reflect not only the nesting level of `let` constructs, but of `fun` constructs too, with a special processing of newly generated Skolem constructors at these occurrences. Now we can very easily enforce our scope restriction: all we have to do is prevent Skolem constructors from being unified with a type variable with a higher binding level.

3.2 Skolem constructors are parameterized

Internally, a Skolem constructor is represented as follows:

```
| TSkolem of int * typ list
```

The `int` is simply a unique stamp used to distinguish different constructors. Then each Skolem constructor carries a `typ list`. Since this fact is not crucial to the understanding of our system, but makes it more complex, it has been overlooked until now.

Every type is of the form

$$\forall \alpha_1, \dots, \alpha_n \; \exists \gamma_1, \dots, \gamma_p . \; t$$

where t is a type expression in which the α_i and γ_i may appear.

When we take an instance of such a type (for instance, in order to conduct pattern matching on a value of this type) we create p new Skolem constructors, which are functions of $\alpha_1, \dots \alpha_n$. The resulting type is of the form

$$\forall \alpha_1, \dots, \alpha_n . \; t \; [\gamma_i / \kappa_i(\alpha_1, \dots \alpha_n)]$$

For instance, let us consider this:

```
#type 'a t = K of 'b * ('b -> 'a);;
Type t defined.
#let x = K (3, succ)
#in let (K (a, b)) = x
#in b a;;
- : int = 4
```

Here `a`'s type is a Skolem constructor with one parameter, `int` (because `x` was of type `int t`).

We then set the following rule : when unifying two Skolems constructors, we must also unify their parameters. For instance, unifying $\kappa(\alpha_1, \dots \alpha_k)$ with $\kappa(\beta_1, \dots \beta_k)$ requires unifying α_i with β_i for all i.

It is not at all obvious that these parameters are necessary to ensure the safety of the type system. Here is an example that demonstrates it. It is the output of a hypothetical version of Caml-Light with non-parameterized Skolem constructors.

[1]Although `match` and `let` should behave the same from typing point of view, since there is no `match` node in Caml-Light abstract syntax trees, a `match` is typed as the application of an anonymous `fun`.

```
#type 'a crash = C of ('a -> 'u) * ('u -> 'a);;
Type crash defined.
#let io = C ((fun x -> x), (fun x -> x));;
io : 'a crash = C (<fun>, <fun>)
#let crash =
#  let (C (i, o)) = io in
#  fun x -> o (i x)
#;;
crash : 'a -> 'b = <fun>
#let x = crash 1
#  in x true;;
Bus error
```

When we write `let (C (i, o)) = io in ...`, `i` and `o` are generalized separately: `i` is given type `'a -> ?c` and `o` type `?c -> 'b`. Therefore, `crash` is thought to be of type `'a -> 'b`, which is incorrect. It is then easy to produce a fatal error: the typechecker thinks `x` is of type `'b`, so `x true` is thought to be valid. Now let us have a look at the output of our compiler:

```
#let crash =
#  let (C (i, o)) = io in
#  fun x -> o (i x)
#;;
crash : 'a -> 'a = <fun>
#let x = crash 1
#  in x true;;
> Toplevel input:
>  in x true;;
>      ^
> Expression of type int
> cannot be used with type 'a -> 'b
```

This time, when evaluating the first `let` clause, the Skolem constructor we create is parameterized. It has one parameter, which is the `'a` variable in `'a crash`. Therefore, after we generalize `i` and `o`, `i` has type `'a -> ?c('a)` and `o` type `?c('b) -> 'b`. When we write `fun x -> o (i x)`, the result type of `i` is unified with the domain type of `o`. When unifying two Skolem constructors, we also unify their parameters; hence `'a` is unified with `'b`, and `crash` is found to have type `'a -> 'a`, which is correct.

3.3 Summary of changes

Here is a short summary of the changes made to the source code. Since some functions changed calling syntax, changes also had to be made every time one of these functions was called. Those unimportant changes have been omitted in the list below.

- File `globals.ml`:
  Type `typ_desc` has been extended with two new constructors, `TExist` and `TSkolem`. The former represents an existentially quantified variable in a type scheme; the latter represents a Skolem constructor, along with its unique tag and its parameter list.

```
  | TExist
  | TSkolem of int * typ list
```

- File `types.ml`:
  Functions `type_instance`, `type_pair_instance`, etc. have undergone a complete rewrite. They now take one more parameter, of type `unit -> typ * typ list`. It is a function that describes what to do when an existential type variable (`TExist`) is encountered. Possible behaviors in such a case are: replace it with a Skolem constructor (used in pattern matching), replace it with a normal type variable (used when constructing a value with an existential type), or raise an exception.

  Function `unify` has been modified in order to handle Skolem constructors. (They can only be unified with themselves, or with a type variable that has a greater binding level). Also, unifying two Skolem constructors implies unifying their parameters.

- File `typing.ml`:
  Function `type_of_type_expression` has been modified in order to handle type declaration with existential variables.

  Function `tpat`, which does the pattern matching job, has been modified. The two interesting cases are records (`Zrecordpat`) and constructors (`Zconstruct1pat`). In both cases, existential variables are instantiated into new Skolems contructors with the proper parameter list.

  Function `type_expr`, the heart of the typing system, has been modified. When constructing values (`Zconstruct1` or `Zrecord`), existential variables are instantiated into normal variables.

  When typechecking a `fun` construct (`Zfunction`), the binding level is incremented. This was not necessary in regular Caml-Light, since no generalization is performed after such constructs. It is now, in order to prevent Skolem constructors to escape out of their scope.

  Also, the way `let` constructs (`Zlet`) are handled has been changed slightly. Otherwise, since the declaration part is one level deeper than the body of the `let`, we would get "Skolems out of scope" errors. We devised a little kluge to avoid this.

- Files `pr_type.ml` and `pr_type.mli`:
  Some functions have been added and others modified in order to properly display existential variables.

- File `ty_error.ml`:
  Function `skolem_going_up_err` was added in order to produce the "existential variable out of scope" error message.

- File `ty_intf.ml`: Added the definition of `check_default_interface` (cf. modifications to file `compiler.ml`).

- File `compiler.ml`:
  Calls to `check_default_interface` were added in order to prevent existential variables from being exported out of a module.

- File `pr_value.ml`:
  Functions `print_val` and `print_concrete_type` have been modified in order to display the `<abstract>` keyword when displaying a value whose type is a Skolem constructor.

- File `pr_type.ml`: Added code for printing existential type variables.
- A few other files were affected by minor changes.

Acknowledgements

We would like to express our thanks to Konstantin Läufer, Chetan Murthy and Didier Rémy for interesting discussions.

References

[1] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Research report 56, DEC Systems Research Center, 1990.

[2] Konstantin Laüfer. *Polymorphic type inference and abstract data types.* PhD thesis, New York University, 1992.

[3] Konstantin Laüfer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the 1992 workshop on ML and its applications*, 1992.

[4] Xavier Leroy. *The Caml Light reference manual.* INRIA, 1993. Included in the Caml Light distribution.