Dynamics in ML

Xavier Leroy
Ecole Normale Supérieure*

Michel Mauny
INRIA Rocquencourt*

Abstract

Objects with dynamic types allow the integration of operations that essentially require run-time type-checking into statically-typed languages. This paper presents two extensions of the ML language with dynamics, based on what has been done in the CAML implementation of ML, and discusses their usefulness. The main novelty of this work is the combination of dynamics with polymorphism.

1 Introduction

Static typing (compile-time enforcement of the typing rules for a programming language) is generally preferred over dynamic typing (the production of run-time tests to check the rules), since static typing reports type violations earlier, and allows for generation of more efficient code. However, one has to revert to dynamic typing for programs that cannot be recognized type-safe at compile-time. This situation often reveals weaknesses of the type system used. Dynamic typing could be avoided, then, by employing more advanced type systems. For instance, it had long been believed that generic functions (functions that can be applied to arguments of different types) can only be supported by dynamically-typed languages, such as Lisp, until the advent of polymorphic type disciplines, such as the one of ML, that permit static typing of such functions.

In contrast, there are programming situations that seem to require dynamic typing in an essential way. A first example is the `eval` function (and similar meta-level operations), that takes a character string, evaluates it as an expression of the language, and returns its value. The type of the returned value cannot be known at compile-time, since it depends on the expression given as argument. Another example is structured input/output. Some runtime systems provide an `extern` primitive, that takes an object of any type, and efficiently outputs a low-level representation of the object to persistent storage. The object can be read back later on, possibly in another process, by the `intern` primitive. The `extern` function can easily be typed in a polymorphic type system; but this is not the case for the `intern` function, since the type of its result depends on the contents of the file being read. In order to guarantee type safety, it is clear that the values returned by `eval` or by `intern`

*Authors' address: INRIA Rocquencourt, projet Formel, B.P. 105, 78153 Le Chesnay, France. Xavier.Leroy, Michel.Mauny@inria.fr

must carry some type information at run-time, and that this type information must be dynamically checked against the type expected by the context.

As demonstrated above, dynamic typing cannot be avoided for a few highly specific functions. But we would like to retain static typing for the huge majority of functions that can be typechecked at compile-time. What we need is a way to embed dynamic typechecking within a statically-typed language. The concept of *objects with dynamic types* (or *dynamics*, for short) is an elegant answer to this need. A dynamic is a pair of a value v and a type expression τ, such that value v has type τ. From the standpoint of static typing, all dynamics belong to the built-in type `dyn`. Type `dyn` represents those values that are self-described as far as types are concerned; that is, those values on which run-time type checking is possible.

Continuing the examples above, function `eval` naturally returns dynamics, so its static type is `string` $\rightarrow$ `dyn`. Similarly, `intern` has type `io_channel` $\rightarrow$ `dyn`, and the `extern` function will be made to accept arguments of type `dyn` only, since the external representation of an object should now include its type.

Two constructs are provided to communicate between type `dyn` and the other types in the language. One construct creates dynamics by taking an object of any type and pairing it with its static type. The other construct checks the internal type of a dynamic against some static type τ, and, in case of success, gives access to the internal value of the dynamic with type τ.

In this paper, we consider the integration of dynamics, as described above, into the ML language [12]. The main novelty of this work is the combination of dynamics with a polymorphic type discipline. This combination raises interesting issues that have not been addressed yet. The main published references on dynamics have only considered first-order types [1], or first-order types with subtyping [4]. Polymorphism gets mentioned in [2], but very briefly and very informally. An early, unpublished work by Mycroft [13] is said to consider the extension of ML with dynamics, but we were unable to find this draft.

The two extensions of ML with dynamics we present here are not mere proposals. The simpler one has been fully integrated into the CAML system [18], the ML implementation developed at INRIA, for more than two years. It has grown to the point of stability where dynamics are used inside the CAML system. The second, more ambitious extension was also extensively prototyped in CAML. This practical experience enables us to discuss the main implementation issues involved by dynamics. It also gives some hints on the practical usefulness of dynamics in an ML system, both for user-level programming and system-level programming.

The remainder of this paper is organized as follows. Section 2 presents a first extension of ML with dynamics. After an informal presentation, we formalize typing and evaluation rules for dynamics within a small subset of ML, and discuss type inference and compilation issues. Section 3 extends the system previously described with the ability to destructure dynamics (both the type part and the value part), and rebuild dynamics with the components of the structure. We adapt the typing and evaluation rules of section 2 to this extension. Section 4 discusses the practical usefulness of the two systems, based on some significant uses of dynamics in the CAML environment. Finally, we give a few concluding remarks in section 5.

2 Simple dynamics

This section describes dynamics as they are implemented in CAML release 2.6 and later [18, chapter 8].

2.1 Presentation

The new construct `dynamic` a is provided to create dynamics. This construct evaluates a, and pairs it with (the representation of) the principal type inferred for a. For instance, `dynamic 1` evaluates to $(1, \texttt{int})$, and `dynamic true` to $(\texttt{true}, \texttt{bool})$. In any case, the expression `dynamic` a is of type `dyn`, without any mention of the internal type of the dynamic.

Objects with polymorphic types can be put in dynamics, provided their type is *closed*: none of the type variables free in their type should be free in the current typing environment. For instance, `dynamic(function x` $\rightarrow$ `x)` is perfectly legal, since the identity function has type $\alpha \rightarrow \alpha$, and α is a fresh type variable, that does not appear anywhere else. The resulting dynamic value is (`function x` $\rightarrow$ `x`, $\forall\alpha.\ \alpha \rightarrow \alpha$). The explicit quantification over α emphasizes the fact that the internal type of the dynamic is closed, and suggests that the internal value is really polymorphic: it will be possible to use it with several types.

On the other hand, `function x` $\rightarrow$ `dynamic x` is rejected: `dynamic x` is typed in the environment `x` : α, where `x` does not have a closed type. In this case, it is problematic to determine at compile-time the exact type of the object put into the dynamic: static typing says it can be any instance of α, that is, any type. To correctly evaluate the function above, the actual type to which α is instantiated would have to be passed at run-time. Since polymorphic functions can be nested arbitrarily, this means that all polymorphic functions, even those that do not build dynamics directly, would have to take type expressions as extra parameters, and correctly propagate these types to the polymorphic functions they call. We would loose one major benefit of static typing: that run-time type information is not needed except inside dynamic objects. Such is the reason why dynamics are required to be created with closed types.

To do anything useful with a dynamic, we must gain access to its internal value, bringing it back to the statically-typed world. A run-time type check is needed at that point to guarantee type safety. This check must ensure that the internal type of the dynamic does match the type expected by the context. This operation is called *coercion* of a dynamic. Coercion is customarily presented as a special syntactic construct (the `typecase` construct in [1]). This construct binds the internal value of the dynamic to some variable. It also handles the case where the run-time type check fails, and another coercion must be attempted, or an exception raised.

In ML, these two aspects, binding and failure handling, are already covered by the pattern-matching mechanism. Hence, instead of providing a separate coercion construct, we integrate dynamic coercion within pattern-matching. Namely, we introduce a new kind of pattern, the dynamic patterns, written `dynamic`$(p : \tau)$. This pattern selects all dynamics whose internal value matches the pattern p, and whose internal type agrees with the type expression τ. For instance, here is a function that

takes a dynamic, and attempts to print a textual representation for it:

```
let print = function
    dynamic(x : int) → print_int x
  | dynamic(s : string) → print_string s
  | dynamic((x, y) : int × int) →
      print_string "("; print_int x; print_string ", ";
      print_int y; print_string ")"
  | x → print_string "?"
```

For type matching, two behaviors can be considered. The first one is to require that the internal type of the dynamic is exactly the same as the expected type, up to a renaming of type variables. The other behavior is to also accept any dynamic whose internal type is more general than the expected type. For instance, `dynamic []`, whose internal type is $\forall\alpha.\ \alpha$ `list`, matches the pattern `dynamic(x : int list)` with the latter behavior, but not with the former. We have retained the latter behavior, since it seems more coherent with the statically-typed part of the ML language (where e.g. the empty list can be used in any context that expects a list of integers).

Type patterns are allowed to require a polymorphic type, as in `dynamic(f :` $\alpha \to \alpha$`)`. This pattern matches any dynamic whose internal type is as general or more general than the type in the pattern (e.g. $\beta \to \beta$, or $\beta \to \gamma$). As a consequence of these semantics, identifier `f` can safely be used with any instance of the type $\alpha \to \alpha$ in the right-hand side of the pattern-matching:

```
function dynamic(f : α → α) → f f
```

The type matching semantics guarantee that `f` will be bound at run-time to a value that belongs to all type instances of the type scheme $\alpha \to \alpha$. This is the only case in ML where a variable bound by a `function` construct can be used with several types inside the function body.

In the example above, the type variable α is not a regular pattern variable such as `f`: it is implicitly quantified universally by the dynamic pattern, and therefore cannot be instantiated during the matching process. For instance, the pattern `dynamic(x :` α `list)` only matches a dynamic of the polymorphic empty list, not any dynamic of any list. As a consequence, a type pattern τ more general than a type pattern τ' will match less dynamics than τ', in contrast with regular ML patterns. This means that in a dynamic matching, the most general type patterns must come first. To catch polymorphic lists as well as integer lists, one must write

```
function dynamic(x : α list) → ...
       | dynamic(x : int list) → ...
```

instead of the more intuitive definition

```
function dynamic(x : int list) → ...
       | dynamic(x : α list) → ...
```

In the latter definition, the second case would never be selected, since the first case also matches dynamics with internal type α `list`.

2.2 Syntax

We now formalize the ideas above, in the context of the core ML language, enriched with pattern-matching and dynamics. The syntax of the language is as follows:

$$\begin{array}{lcl} \tau & ::= & \texttt{int} \parallel \alpha \parallel \tau \to \tau' \parallel \tau \times \tau' \parallel \texttt{dyn} \\ p & ::= & x \parallel i \parallel (p, p') \parallel \texttt{dynamic}(p : \tau) \\ a & ::= & x \parallel i \parallel \texttt{function}\ p_1 \to a_1 \mid \ldots \mid p_n \to a_n \parallel a\ a' \parallel (a, a') \\ & & \parallel \texttt{let}\ p = a\ \texttt{in}\ a' \parallel \texttt{dynamic}\ a \end{array}$$

Type expressions (with typical elements τ, σ) are either atomic types such as the type `int` of integers; type variables α; function types; product types; or the type `dyn` of dynamics.

Patterns (typical elements p,q) are either variables x; integer constants i; pairs of patterns; or dynamic patterns $\texttt{dynamic}(p : \tau)$.

Finally, for expressions (typical elements a,b), we have variables x; integer constants i; functions with pattern matching on the argument; function application; pair construction; the `let` binding with pattern matching on the first expression; and the `dynamic` construct to build dynamics.

2.3 Typechecking

The typing rules for this calculus are given in figure 1. Most of the rules are just those of the core ML language, revised to take pattern-matching into account in the `function` and `let` constructs. Two additional rules present the creation and coercion of dynamics.

The rules define the predicate $E \vdash a : \tau$, meaning "expression a has type τ in the typing environment E". An auxiliary predicate, $\vdash p : \tau \Rightarrow E$, is used, meaning "pattern p has type τ and enriches the type environment by E". Here, E stands for a sequence of typing assumptions of the format $x : \Sigma$, where Σ is a type scheme: a type expression τ with zero, one or more type variables α_k universally quantified.

$$E \quad ::= \quad (x : \forall \alpha_1 \ldots \alpha_n . \tau)^*$$

We write $E(x)$ for the type associated to x in E. If x is bound several times, we consider the rightmost binding only. We write $FV(\tau)$ for the free variables of type expression τ, and $FV(E)$ for the union of the free variables of all type schemes in E. Finally, $Clos(\tau, V)$ stands for the closure of type τ with respect to those type variables not in the set V. It is defined by:

$$Clos(\tau, V) = \forall \alpha_1 \ldots \alpha_n . \tau$$

where $\{\alpha_1, \ldots, \alpha_n\}$ is $FV(\tau) \setminus V$. The $Clos$ operator is extended pointwise to type environments.

The only new rules are rule 7, that deals with dynamic creation, and rule 11, that deals with dynamic coercion. Rule 7 says that the expression `dynamic` a has type `dyn`, provided that a has a closed type τ: none of the free variables of τ are free in the current typing environment E.

$$(1)\quad E \vdash i : \texttt{int} \qquad (2)\quad \frac{E(x) = \forall\alpha_1 \ldots \alpha_n.\tau}{E \vdash x : \tau[\alpha_k \leftarrow \tau_k]}$$

$$(3)\quad \frac{\vdash p_k : \sigma \Rightarrow E_k \quad E, E_k \vdash a_k : \tau}{E \vdash \texttt{function}\ p_1 \rightarrow a_1 \mid \ldots \mid p_n \rightarrow a_n : \sigma \rightarrow \tau}$$

$$(4)\quad \frac{E \vdash a : \sigma \rightarrow \tau \quad E \vdash b : \sigma}{E \vdash a\ b : \tau} \qquad (5)\quad \frac{E \vdash a : \sigma \quad E \vdash b : \tau}{E \vdash (a, b) : \sigma \times \tau}$$

$$(6)\quad \frac{E \vdash a : \sigma \quad \vdash p : \sigma \Rightarrow E' \quad E, Clos(E', FV(E)) \vdash b : \tau}{E \vdash \texttt{let}\ p = a\ \texttt{in}\ b : \tau}$$

$$(7)\quad \frac{E \vdash a : \sigma \quad FV(\sigma) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ a : \texttt{dyn}}$$

$$(8)\quad \vdash x : \tau \Rightarrow x : \tau \qquad (9)\quad \vdash i : \texttt{int} \Rightarrow \epsilon$$

$$(10)\quad \frac{\vdash p : \tau \Rightarrow E \quad \vdash p' : \tau' \Rightarrow E'}{\vdash (p, p') : \tau \times \tau' \Rightarrow E, E'} \qquad (11)\quad \frac{\vdash p : \tau \Rightarrow E}{\vdash \texttt{dynamic}(p : \tau) : \texttt{dyn} \Rightarrow Clos(E, \emptyset)}$$

Figure 1: Typing rules

Rule 11 says that the pattern `dynamic`$(p : \tau)$ matches values of type `dyn`, provided p matches values of type τ. Assume p binds variables $x_1 \ldots x_n$ to values of types $\tau_1 \ldots \tau_n$. Then, `dynamic`$(p : \tau)$ binds the same variables to the same values. As described above, all type variables free in $\tau_1 \ldots \tau_n$ can be generalized. Hence, we take that `dynamic`$(p : \tau)$ binds $x_1 \ldots x_n$ to values of types $Clos(\tau_1, \emptyset) \ldots Clos(\tau_n, \emptyset)$.

2.4 Type inference

The type system presented above enjoys the principal type property, just as the one of ML. The principal type is computed by a trivial extension of the Damas-Milner type inference algorithm [5] to handle the `dynamic` construct and dynamic patterns. For the `dynamic` a construct, the only difficulty is to ensure that the type of a is closed. It would not be correct to infer the most general type τ for a, and fail immediately if some variables in τ are free in the current typing environment: these variables may become instantiated to monomorphic types later. Consider the expression (`function x` $\rightarrow$ `dynamic x`) `1`. Assuming the function part of the application is typed before the argument, the function is given type $\alpha \rightarrow$ `dyn`, and `dynamic x` appears to build a dynamic with non-closed type α. But when the application is typed, α gets instantiated to `int`, and we know that the dynamic is created with internal type `int`. Hence, the closedness check must be delayed to the end of type inference. The CAML implementation proceeds as follows: when typing `dynamic` a, all type variables that are free in the inferred type for a and in the current

$$(12)\quad e \vdash x \Rightarrow e(x) \qquad\qquad (13)\quad e \vdash i \Rightarrow i$$

$$(14)\quad e \vdash \texttt{function}\ \ldots \mid p_k \to a_k \mid \ldots \Rightarrow [e, \ldots \mid p_k \to a_k \mid \ldots]$$

$$(15)\quad \frac{e \vdash a \Rightarrow [e', \ldots \mid p_k \to a_k \mid \ldots] \quad e \vdash b \Rightarrow v \quad \vdash v < p_k \Rightarrow e'' \quad e', e'' \vdash a_k \Rightarrow w \quad k \text{ minimal}}{e \vdash a\ b \Rightarrow w}$$

$$(16)\quad \frac{e \vdash a \Rightarrow v \quad e \vdash b \Rightarrow w}{e \vdash (a, b) \Rightarrow (v, w)} \qquad (17)\quad \frac{e \vdash a \Rightarrow v \quad \vdash v < p \Rightarrow e' \quad e, e' \vdash b \Rightarrow w}{e \vdash \texttt{let}\ p = a\ \texttt{in}\ b \Rightarrow w}$$

$$(18)\quad \frac{e \vdash a \Rightarrow v \quad Type(a) = \tau}{e \vdash \texttt{dynamic}\ a \Rightarrow \texttt{dynamic}(v : \tau)}$$

$$(19)\quad \vdash v < x \Rightarrow x \leftarrow v \qquad\qquad (20)\quad \frac{\vdash v < p \Rightarrow e \quad \vdash v' < p' \Rightarrow e'}{\vdash (v, v') < (p, p') \Rightarrow e, e'}$$

$$(21)\quad \vdash i < i \Rightarrow \epsilon \qquad\qquad (22)\quad \frac{\vdash v < p \Rightarrow e \quad \theta\tau = \sigma}{\vdash \texttt{dynamic}(v : \tau) < \texttt{dynamic}(p : \sigma) \Rightarrow e}$$

Figure 2: Evaluation rules

typing environment are collected in a list, and prevented from being generalized. At the end of typechecking, all type variables in the list must be instantiated by ground types.

For dynamic patterns $\texttt{dynamic}(p : \tau)$, the expected type τ is given explicitly in the pattern, so there is actually nothing to infer. We just check that the pattern p is of type τ, and record the (polymorphic) types of the variables bound by p. We have considered inferring τ from the pattern p and the right-hand side a of the pattern-matching, but this seems quite difficult, since variables bound by p can be used with several different types in a.

2.4.1 Evaluation

We now give call-by-value operational semantics for our calculus. Expressions are mapped to values, that is, terms with the following syntax:

$$\begin{array}{rcl} v & ::= & i \parallel (v, v') \parallel \texttt{dynamic}(v : \tau) \parallel [e, p_1 \to a_1 \mid \ldots \mid p_n \to a_n] \\ e & ::= & (x \leftarrow v)^* \end{array}$$

A value is either an integer i; a pair of values; a dynamic value $\texttt{dynamic}(v : \tau)$ (a pair of a value v and a type expression τ); or a closure $[e, p_1 \to a_1 \mid \ldots \mid p_n \to a_n]$ of function body $p_1 \to a_1 \mid \ldots \mid p_n \to a_n$ by evaluation environment e. Evaluation environments map identifiers to values in the same way as typing environments map identifiers to types. For dynamic values, all type variables in the type part are

considered universally quantified there; hence, two dynamic values are identified up to a renaming of their type variables.

The evaluation rules are given in figure 2. They closely follow the structure of the typing rules. A first set of rules define the predicate $e \vdash a \Rightarrow v$, meaning "expression a evaluates to value v in environment e". The remaining rules define the auxiliary predicate $\vdash v < p \Rightarrow e$, meaning "value v matches pattern p, binding variables in p to values as described by e".

Since most rules are classical, we only detail the two rules dealing with dynamics. Rule 18 expresses that evaluating `dynamic` a amounts to evaluating a, and pairing its value with the static type of a. The type of a is not mentioned in the expression `dynamic` a, so there are some technicalities involved in defining precisely what it is. We assume all expressions a considered here are subterms of a given closed, well-typed term a_0 (the program). Let $\mathcal{D}$ be the principal type derivation for a_0 (the one that is built by the type inference algorithm). For each subterm a of a_0, $Type(a)$ is defined as the type given to a in $\mathcal{D}$. (In practice, dynamic expressions are annotated with their types during typing.)

Rule 22 defines the semantics of pattern matching over dynamics. The internal type τ of the dynamic is required to be more general than the type σ expected by the pattern: there must exist a substitution θ of types for type variables such that $\theta\tau$ is σ. The internal value of the dynamic is recursively matched against the value part of the dynamic pattern.

2.5 Compilation

In the current CAML implementation, internal types of dynamics are represented by the following term-like structure:

```
type gtype = Gvartype of int
           | Gconsttype of int × gtype list
```

Type constructors are identified by unique stamps instead of names to correctly handle type redefinition. Type variables are also encoded as integers. The code generated for `dynamic` a simply pairs the value of a with the structured constant representing $Type(a)$ as a `gtype`. For pattern matching, CAML provides a library function `ge_gtype`, that takes two types and tests whether the first one is more general than the second one. The code generated for pattern matching on dynamics simply calls `ge_gtype` with the internal type of the dynamic, and the expected type (again, a structured constant of type `gtype`). Only when `ge_gtype` returns true is the sequence of tests matching the internal value against the pattern entered. Those tests were compiled assuming that the value being tested belongs to the expected type for the dynamic; therefore, it would be incorrect to match the internal value first, and then the internal type.

Little effort went into making run-time type tests faster. We have not yet encountered CAML programs that need to perform dynamic coercions inside tight loops. In case coercion speed becomes an important issue, we could first switch to the following representation for internal types of dynamics:

```
type gtype = Gvartype of gtype option ref
```

```
  | Gconsttype of int × gtype list
```

This representation makes it possible to perform instantiations by physical modifications on the type, which is more efficient than recording them separately as a substitution. (These physical modifications are undone at the end of the matching.)

Then, we could perform partial evaluation on the `ge_gtype` predicate, since its second argument is always known at compile-time. Conventional pattern-matching compilation techniques [14] do not apply directly, however, since they consist in specializing term-matching predicates on their first argument (the more general term), not on the second one (the less general term). Specializing a matching predicate such as `ge_gtype` on its second argument turns out to be just as hard as the more general problem of specializing a unification predicate on one of its arguments. The latter problem has been extensively studied in the context of Prolog compilation. A popular solution is the Warren Abstract Machine and its compilation scheme [17, 9]. Most of the techniques developed there apply to our problem. We shall detail this issue at the end of section 3.6.

3 Non-closed types in dynamic patterns

This section presents an extension of the system presented above that makes it possible to match dynamic values against dynamic patterns with incomplete type information. This enables destructuring dynamics without specifying their exact type.

3.1 Presentation

With the previous system, the internal value of a dynamic can only be extracted with a fixed type. This turns out to be insufficient in some cases. Let us continue the `print` example of section 2.1. For product types, we would like to have a single case that matches all dynamics of pairs, prints the parentheses and comma, and recursively calls the `print` function to print the two components of the pair. This cannot be done with the system above: the pattern `dynamic`$((\texttt{x}, \texttt{y}) : \alpha \times \beta)$ will only match dynamics whose internal type is at least as general as $\forall\alpha\forall\beta.\, \alpha \times \beta$, definitely not all dynamics whose internal type is a pair type. What we need is to have type variables in dynamic patterns that are not universally quantified, but rather existentially quantified, so that they can be bound to the corresponding parts of the internal type of the dynamic.

We now give a more complete version of the `print` function, with explicit universal and existential quantification for type variables in dynamic patterns. We will use it as a running example in this section.

```
type fun_arg = Arg of string in
let rec print = function
    dynamic(i : int) →                                              (1)
        print_int i
  | dynamic(s : string) →                                           (2)
        print_string "\""; print_string s; print_string "\""
```

```
| ∃α.∃β.dynamic((x, y) : α × β)  →                                    (3)
      print_string "("; print(dynamic x); print_string ", ";
      print(dynamic y); print_string ")"
| ∃α.dynamic([] : α list)  →                                          (4)
      print_string "[]"
| ∃α.dynamic(x :: l : α list)  →                                     (5)
      print(dynamic x); print_string " :: "; print(dynamic l)
| ∀α.dynamic(f : α → α)  →                                           (6)
      print_string "function x → x"
| ∃α.∀β.dynamic(f : α → β)  →                                        (7)
      print_string "function x → ⊥"
| ∀α.∃β.dynamic(f : α → β)  →                                        (8)
      let s = gensym() in
        print_string "function "; print_string s;
        print_string " → "; print(dynamic(f (Arg s)))
| dynamic(Arg(s) : fun_arg)  →                                       (9)
      print_string s
| ∃α.∃β.dynamic(f : α → β)  →                                        (10)
      print_string "function x → ..."
| d  →                                                               (11)
      print_string "?"
```

Typing existential quantification

Let us detail first how these existentially quantified type variables behave when typing the right-hand side of the pattern-matching. Such a variable α can be bound to any actual type at run-time. Hence, at compile-time, we should make no assumptions about type α, and treat it as an abstract type. That is, type α does not match any type except itself; and type α must not escape the scope of the pattern-matching that binds it: α is not allowed to be free in the type of the returned value. As a consequence, the following two functions are rejected:

```
function ∃α. dynamic(x : α)  →  x = 1
function ∃α. dynamic(x : α)  →  x
```

while this one is perfectly legal:

```
function ∃α. dynamic((f, x) : (α → int) × α)  →  f x
```

and can be applied to `dynamic(succ, 2)` as well as to `dynamic(int_of_string, "3")`.

There is one important difference between existentially bound type variables and abstract types: the actual type bound to such a type variable is available at run-time. Given an object a whose static type contains a variable α existentially bound, it is possible to build a dynamic from this object. The internal type of the dynamic will be the "true" type for a: its static type where the binding of α has been performed. Cases (3) and (5) in the `print` function illustrates this feature: when the matching with $\exists\alpha$. `dynamic(x :: l : α list)` succeeds, two dynamics are created, `dynamic x` with internal type the type τ bound to α; and `dynamic l` with internal type τ `list`. This transforms a dynamic of a non-empty list into the dynamic of its head, and the dynamic of its tail, thus allowing recursion on the list.

Mixed quantifications

Existentially quantified variables can be freely mixed with universally quantified variables inside type patterns. Then, the semantics of the matching depends on the relative order in which these variables are quantified. This is illustrated by cases (7) and (8) in the `print` example — two modest attempts at printing functional values.

In case (7), the pattern is $\exists\alpha.\forall\beta.$`dynamic`$($`f` $: \alpha \to \beta)$. Since α is bound before β, variable α only matches type expressions that do not depend on β. For instance, a dynamic with internal type $\gamma \to \gamma$ is rejected. The functions selected by the pattern above are exactly those returning a value of type β for all β. Since no such value exists in ML, the selected functions never terminate (or always raise an exception), hence they are printed as `function x` $\to \bot$.

In case (8), the pattern is $\forall\alpha.\exists\beta.$`dynamic` $($`f` $: \alpha \to \beta)$. Here, β is bound after α; hence β can be instantiated by type expressions containing α. For instance, this pattern matches a dynamic with type $\gamma \to \gamma$ `list`, binding β to α `list`. This pattern catches a class of functions that operate uniformly on arguments of any type. These functions cannot test or destructure their argument, but only put it in data structures or in closures. Therefore, if we apply such a function to a symbolic name `x`, and recursively print the result, we get a representation of the function body, with `x` standing for the function parameter[1]. (Actually, the printed function is extensionally equivalent to the original function, assuming there are no side-effects.)

In the presence of mixed quantification, the rules for typing the right-hand side of pattern-matchings outlined above have to be strengthened: it is not always correct to treat an existentially quantified type variable as a new abstract atomic type. Consider:

`function` $\forall\alpha.\exists\beta.$`dynamic`$($`f` $: \alpha \to \beta)$ $\to$ `f 1 = f true`

Assuming `f` $: \forall\alpha.\ \alpha \to \beta$, the expression `f 1 = f true` typechecks, since both applications of `f` have type β. Yet, when applying the function above to `dynamic` (`function x` $\to$ `x`), the matching succeeds, `f 1` evaluates to `1`, `f true` evaluates to `true`, and we end up comparing `1` with `true` — a run-time type violation. Since the actual value of β is allowed to depend on α, static typechecking has to assume that β does depend over α, and treat two occurrences of β corresponding to different instantiations of α as incompatible.

This is achieved by taking β in the right-hand side of the matching to be a type constructor parameterized by α. To avoid confusion, we shall write $\overline{\beta}$ for the type constructor associated to type variable β. Therefore, we now assume `f` $: \forall\alpha.\ \alpha \to \overline{\beta}(\alpha)$ for the typing of `f 1 = f true`, and this leads to a static type error, since the two sides of the equal sign have incompatible types $\overline{\beta}$(`int`) and $\overline{\beta}$(`bool`). However, `f 1 = f 2` is well-typed, since both sides have type $\overline{\beta}$(`int`). The general rule is: for the purpose of typing the right-hand side of a pattern-matching, existentially

[1] To avoid any confusion between the formal parameter and constants mentioned in the function body, formal parameters are represented by a local type `fun_arg = Arg of string`. This ensures that the given function cannot create any terms of type `fun_arg`, unless it is the `print` function itself. Fortunately, the self-application `print(dynamic print)` selects case (10) of the definition.

quantified type variables β are replaced by the type expression $\overline{\beta}(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1 \ldots \alpha_n$ is the list of those type variables that are universally quantified before β in the pattern. This transformation is known in logic as Skolemization.

Multiple dynamic matching

Type variables are quantified at the beginning of each case of the pattern-matching, not inside each dynamic pattern. This makes no difference for universally quantified variables. However, existentially quantified variables can be shared among several dynamic patterns, expressing sharing constraints between the internal types of several dynamics. For instance, the "dynamic function application" example of [1] can be written as:

$$\texttt{function}\ \exists\alpha.\exists\beta.\ (\texttt{dynamic}(\texttt{f} : \alpha \to \beta),\ \texttt{dynamic}(\texttt{x} : \alpha)) \ \to \\ \texttt{dynamic}(\texttt{f x})$$

This function takes a pair of two dynamics, applies the first one (which should contain a function) to the second one, and returns the result as a dynamic. It ensures that the type of the argument is compatible with the domain type of the function.

Type variables can be shared among two dynamic patterns of the same matching; but we shall prohibit sharing between patterns belonging to different matchings (curried dynamic matching). In other terms, all cases in a pattern matching are required to be closed: all type variables contained in dynamic patterns should be quantified at the beginning of the corresponding matching. For instance, it is not possible to write the dynamic apply function in the following way (as it originally appears in [1]):

$$\texttt{function}\ \exists\alpha.\exists\beta.\ \texttt{dynamic}(\texttt{f} : \alpha \to \beta) \ \to\ \texttt{function dynamic}(\texttt{x} : \alpha) \ \to \\ \texttt{dynamic}(\texttt{f x})$$

This violates the requirement above, since α is bound by the outermost matching, and mentioned in the innermost one. The reasons for this restriction are mostly pragmatic: curried dynamic matching, in conjunction with polymorphic dynamics, can require postponing some type matching in an outer dynamic matching until an inner dynamic matching is performed. Our closedness condition on pattern matching cases rules out these nasty situations, without significantly reducing the expressive power of the language: curried dynamic application can still be written as

$$\texttt{function df} \ \to\ \texttt{function dx} \ \to\ \texttt{match}\ (\texttt{df}, \texttt{dx})\ \texttt{with}\ \ldots$$

at the expense of a later error detection, in case `df` is not a dynamic of function.

3.2 Syntax

The only syntactic change is the introduction of a sequence of quantifiers in front of each case in pattern matchings.

$$\begin{array}{rcl} a & ::= & \ldots \parallel \texttt{function}\ Q_1 p_1 \to a_1 \mid \ldots \mid Q_n p_n \to a_n \\ Q & ::= & \epsilon \parallel \forall\alpha.Q \parallel \exists\alpha.Q \end{array}$$

$$(23)\quad \frac{Q_k \vdash p_k : \sigma \Rightarrow E_k \quad E, E_k \vdash a_k : \tau \quad FSC(\tau) \cap BV(Q_k) = \emptyset}{\texttt{function}\ Q_1 p_1 \rightarrow a_1 \mid \ldots \mid Q_n p_n \rightarrow a_n : \sigma \rightarrow \tau}$$

$$(24)\quad Q \vdash x : \tau \Rightarrow x : \tau \qquad (25)\quad Q \vdash i : \texttt{int} \Rightarrow \epsilon$$

$$(26)\quad \frac{Q \vdash p : \tau \Rightarrow E \quad Q \vdash p' : \tau' \Rightarrow E'}{Q \vdash (p, p') : \tau \times \tau' \Rightarrow E, E'}$$

$$(27)\quad \frac{FV(\tau) \subseteq BV(Q) \quad Q \vdash p : \tau \Rightarrow E \quad \theta = S(\epsilon, Q)}{Q \vdash \texttt{dynamic}(p : \tau) : \texttt{dyn} \Rightarrow Clos(\theta\tau, \emptyset)}$$

Figure 3: Typing rules with explicit quantification in type patterns

We will always assume that variables are renamed so that quantifier prefixes Q never bind the same variable twice. We write $BV(Q)$ for the set of variables bound by prefix Q.

3.3 Typechecking

We introduce the Skolem constants at the level of types. To each type variable α, we associate the type constructor $\overline{\alpha}$, with variable arity.

$$\tau \quad ::= \quad \ldots \parallel \overline{\alpha}(\tau_1, \ldots, \tau_n)$$

Skolem constants are not permitted in the type part of dynamic patterns, nor in the internal types of dynamic values. We shall write τ^0, σ^0 for type expressions free of Skolem constants. We define $FSC(\tau)$, the free Skolem constants of type τ, as the set of all variables α such that type constructor $\overline{\alpha}$ appears in τ.

The new typing rules for functions and for patterns are shown in figure 3. For each case $Qp \rightarrow a$ in a function definition, the pattern p is typed taking Q into account. The proposition $\vdash p : \sigma \Rightarrow E$ now takes Q as an extra argument, becoming $Q \vdash p : \sigma \Rightarrow E$. The Q prefix is carried unchanged through all rules, and it is used only in the rule for dynamic patterns. There, in the types of all identifiers bound by the pattern, we replace existentially quantified type variables by the corresponding Skolem functions. This is performed by the substitution $\theta = S(\epsilon, Q)$, defined inductively on Q as follows:

$$\begin{aligned} S(\alpha_1 \ldots \alpha_n, \epsilon) &= id \\ S(\alpha_1 \ldots \alpha_n, \forall\alpha.Q) &= S(\alpha_1 \ldots \alpha_n \alpha, Q) \\ S(\alpha_1 \ldots \alpha_n, \exists\alpha.Q) &= [\alpha \leftarrow \overline{\alpha}(\alpha_1, \ldots, \alpha_n)] \circ S(\alpha_1 \ldots \alpha_n, Q) \end{aligned}$$

Typing of action a proceeds as previously. We simply check that the type of τ does not contain any Skolem constants corresponding to variables bound by Q.

$$(28)\quad e \vdash \texttt{int} \Rightarrow \texttt{int} \qquad (29)\quad e \vdash \alpha \Rightarrow \alpha \qquad (30)\quad e \vdash \texttt{dyn} \Rightarrow \texttt{dyn}$$

$$(31)\quad \frac{e \vdash \sigma \Rightarrow \sigma^0 \quad e \vdash \tau \Rightarrow \tau^0}{e \vdash \sigma \to \tau \Rightarrow \sigma^0 \to \tau^0} \qquad (32)\quad \frac{e \vdash \sigma \Rightarrow \sigma^0 \quad e \vdash \tau \Rightarrow \tau^0}{e \vdash \sigma \times \tau \Rightarrow \sigma^0 \times \tau^0}$$

$$(33)\quad \frac{e(\alpha) = [\alpha_1 \ldots \alpha_n].\tau^0 \quad e \vdash \tau_k \Rightarrow \tau_k^0}{e \vdash \overline{\alpha}(\tau_1 \ldots \tau_n) \Rightarrow \tau^0[\alpha_k \leftarrow \tau_k^0]}$$

$$(34)\quad \frac{e \vdash \tau \Rightarrow \tau^0 \quad e \vdash a \Rightarrow v}{e \vdash \texttt{dynamic}\ a \Rightarrow \texttt{dynamic}(v : \tau^0)}$$

$$(35)\quad \frac{\begin{array}{c} e \vdash a \Rightarrow [e_1, \ldots \mid Q_k p_k \to a_k \mid \ldots] \quad e \vdash b \Rightarrow v \quad k \text{ minimal} \\ Q_k \vdash v < p_k \Rightarrow e_2 \,;\, \Gamma \quad Solve(Q_k, \Gamma) = e_3' \quad e_1, e_2, e_3 \vdash a_k \Rightarrow w \end{array}}{e \vdash a\ b \Rightarrow w}$$

$$(36)\quad Q \vdash v < x \Rightarrow x \leftarrow v \,;\, \epsilon \qquad (38)\quad \frac{Q \vdash v < p \Rightarrow e \,;\, \Gamma \quad Q \vdash v' < p' \Rightarrow e' \,;\, \Gamma'}{Q \vdash (v, v') < (p, p') \Rightarrow e, e' \,;\, \Gamma, \Gamma'}$$

$$(37)\quad Q \vdash i < i \Rightarrow \epsilon \,;\, \epsilon \qquad (39)\quad \frac{Q \vdash v < p \Rightarrow e \,;\, \Gamma \quad FV(\tau) \cap BV(Q) = \emptyset}{Q \vdash \texttt{dynamic}(v : \tau) < \texttt{dynamic}(p : \sigma) \Rightarrow e \,;\, \Gamma, \tau = \sigma}$$

Figure 4: Evaluation rules with explicit quantification in type patterns

3.4 Evaluation

The introduction of existential type variables in dynamic patterns significantly complicates the semantics of the language, both for dynamic creation, and for dynamic matching. The modified evaluation rules are shown in figure 4.

For dynamic creation (rule 34), the evaluation of `dynamic` a now has to transform the static type τ inferred for a before pairing it with the value of a. Skolem constants representing existentially bound type variables are replaced by the actual types bound to these variables, properly instantiated. These bindings of type variables are recorded in the evaluation environment e. Hence the new syntax for evaluation environments:

$$e \quad ::= \quad (x \leftarrow v \parallel \alpha \leftarrow [\alpha_1, \ldots, \alpha_n]\tau^0)^*$$

Since existential type variables may depend upon universal variables, existential variables are actually bound to a type context (a type expression with holes) instead of a simple type expression. We write type contexts as $[\alpha_1, \ldots, \alpha_n]\tau^0$, where type variables $\alpha_1 \ldots \alpha_n$ are names for the holes. Rules 28–33 define the evaluation relation on types $e \vdash \tau \Rightarrow \tau^0$, mapping a type expression τ to a type expression τ^0 without Skolem constants.

For dynamic matching during function application (rule 35), it is not possible anymore to perform dynamic type matching separately for each dynamic pattern, since patterns may share existentially quantified variables. Therefore, all dynamic type constraints are collected first, as a set of equations $\tau = \sigma$, where τ is the internal type of a dynamic, and σ a type pattern. The pattern-matching predicate becomes $Q \vdash v < p \Rightarrow e\,;\Gamma$, where Γ is the sequence of equations between types described above, and Q is the quantifier prefix for the matching (rules 36–39). The Q prefix is used in rule 39 to rename the internal types of dynamics, if necessary, so that their free type variables are not bound by Q. In a second phase, the function *Solve* is called to resolve the equations on types Γ, taking prefix Q into account. The precise definition of *Solve* is postponed to the next section. When the type matching succeeds, *Solve* returns the correct bindings for existentially quantified type variables. Then, evaluation of the right-hand side of the matching proceeds as usual.

3.5 Unification

The run-time matching between type patterns and internal types of dynamics amounts to a certain kind of unification problem, called *unification under a prefix*. This problem is studied extensively in [11], though in the very general setting of higher-order unification, while we only deal with first-order terms here. The first-order problem also appears in [10]. In our case, the problem consists in checking the validity of propositions of the format

$$q_1\alpha_1 \ldots q_n\alpha_n.\ \sigma = \tau,$$

where the q_k are either universal or existential quantifiers, and σ, τ are first-order terms of a free algebra. Unification under mixed prefix generalizes the well-known matching problem ("given two terms σ, τ, find a substitution θ such that $\theta\sigma = \tau$") and the unification problem ("given two terms σ, τ, find a substitution θ such that $\theta\sigma = \theta\tau$"): writing $\alpha_1 \ldots \alpha_n$ for the variables of σ, and $\beta_1 \ldots \beta_n$ for the variables of τ, the matching problem is equivalent to

$$\forall\beta_1 \ldots \forall\beta_n \exists\alpha_1 \ldots \exists\alpha_n.\ \sigma = \tau,$$

and the unification problem to

$$\exists\beta_1 \ldots \exists\beta_n \exists\alpha_1 \ldots \exists\alpha_n.\ \sigma = \tau.$$

For the purpose of dynamic matching, we not only want to know whether the problem $Q.\ \sigma = \tau$ is satisfiable (Q is a quantifier prefix), but also to find minimal assignments for the variables existentially quantified in Q that satisfy the proposition. From now on, we shall treat variables universally bound in Q as constants. That is, we add such variables as term constructors with arity zero to the initial signature (`int` and `dyn` of arity zero, $\rightarrow$ and $\times$ of arity two).

Definition 1 *A substitution θ is a Q-substitution iff for all variables α, all constants β contained in the term $\theta\alpha$ are bound before α in prefix Q.*

Definition 2 *A substitution θ is a Q-unifier of σ and τ iff $\theta\sigma = \theta\tau$, and θ is a Q-substitution. If such a substitution exists, σ and τ are said to be Q-unifiable.*

Proposition 1 *The proposition $Q.\ \sigma = \tau$ is satisfiable if and only if σ and τ are Q-unifiable.*

Proposition 2 *Two terms σ and τ are Q-unifiable if and only if σ and τ are unifiable, and their most general unifier is a Q-substitution.*

Proof: The "if" part is obvious. For the "only if" part, let θ be a Q-unifier of σ and τ. Since $\theta\sigma = \theta\tau$, the terms σ and τ are unifiable. Let μ be their most general unifier. Let ϕ be a substitution such that $\theta = \phi \circ \mu$. For all variables α, the constants contained in $\mu\alpha$ are a subset of those contained in $\theta\alpha$. Since θ is a Q-substitution, all constants in $\mu\alpha$ are also bound before α in Q. Hence μ is a Q-substitution. □

This result trivially gives an algorithm to compute the most general Q-unifier of σ and τ: compute the most general unifier of σ and τ, using Robinson's algorithm, and check that it is a Q-substitution.

We can now define the function *Solve* used in evaluation rule 32. It takes a prefix Q and a set Γ of equations $\sigma_1 = \tau_1 \ldots \sigma_n = \tau_n$. Since Q binds the variables in the τ_k only, prefix Q is first completed to bind the variables in the σ_k also. Let $\alpha_1 \ldots \alpha_m$ be the variables in the σ_k. We take $Q' = Q.\exists\alpha_1 \ldots \exists\alpha_m$. (None of the α_k is bound by Q, and they can be instantiated to any type.) Let μ be the most general Q'-unifier of $\sigma_1 \times \ldots \times \sigma_n$ and $\tau_1 \times \ldots \times \tau_n$, as computed by the algorithm above. Substitution μ is transformed into an evaluation environment, by adding bindings for the variables that are existentially quantified in Q. More precisely, we take $Solve(Q, \Gamma)$ to be $s(\mu, \epsilon, Q)$, where s is the run-time counterpart of the Skolemization function S used for static typing in section 3.3:

$$\begin{array}{rcl} s(\mu, \alpha_1 \ldots \alpha_n, \epsilon) & = & \epsilon \\ s(\mu, \alpha_1 \ldots \alpha_n, \forall\alpha.Q) & = & s(\mu, \alpha_1 \ldots \alpha_n\alpha, Q) \\ s(\mu, \alpha_1 \ldots \alpha_n, \exists\alpha.Q) & = & \alpha \leftarrow [\alpha_1, \ldots, \alpha_n]\mu\alpha,\ s(\mu, \alpha_1 \ldots \alpha_n, Q) \end{array}$$

3.6 Compilation

The semantics given above are quite complicated, so it is no surprise their implementation turns out to be delicate. The main difficulty is unification under a prefix Q. Efficient algorithms are available for the regular unification phase. It remains to quickly check that the resulting substitution is a Q-substitution. This check can actually be integrated within the occur check, at little extra cost. The idea is to reflect dependencies by associating ranks (integers) to type variables. Variables bound by Q are statically given ranks $0, \ldots, n$ from left to right. Other variables (i.e. those in the internal types of dynamics) are considered bound at the end of Q, and therefore given rank ∞. When identifying two variables α and β, the resulting variable is given rank $min(rank(\alpha), rank(\beta))$. Then, binding existential variable α to a constructed type τ is legal iff:

1. (occur check) α does not occur in τ
2. (rank check) τ does not contain any universal type variable whose rank is greater than the rank of α.

As in the case of simple dynamics (section 2.5), the easiest way to implement type matching is to call at run-time a unification primitive, with the type pattern (annotated by rank information) as a constant argument. Partial evaluation of the unification primitive on the type pattern is desirable, not only to speed up type matching, but also to provide a cleaner handling of run-time type environments: after specialization, the bindings for the existential type variables could be recorded e.g. on the stack or in registers, as for regular variables; without specialization, the unification primitive would return a data structure containing these bindings, and less efficient code would be generated to access these bindings.

Specializing unification on one of its arguments is not much harder than specializing matching on its second argument (section 2.5). The techniques developed for the Warren Abstract Machine [17, 9] directly apply, with the exception of the extra rank check. For instance, the WAM does not perform occur check for the initial binding of an existential variable, while we have to check ranks even in this case. Another difference is that backtracking is always "shallow", in the WAM terminology, since ML pattern-matching is deterministic. This simplifies the handling of the trail.

During the summer of 1988, the first author integrated a prototype unification compiler in the CAML system, following the ideas above. The CAML pattern-matching compiler was modified to implement unification semantics as well as matching semantics, depending on flags put on the patterns. This low-level mechanism allowed performing unification on some parts of a data structure, and regular pattern-matching on the other parts. Then, dynamic patterns $\texttt{dynamic}(p : \tau)$ were simply expanded after type inference into product patterns $(p, \texttt{repr}(\tau))$, where $\texttt{repr}(\tau)$ is the pattern that matches all internal representations of types matching τ. The pattern $\texttt{repr}(\tau)$ is marked to use unification semantics.

The only missing feature from what we have described above was rank check. At that time, we considered only dynamic patterns where all universal type variables come first, followed by all existential variables. Rank check could have been added with little modifications.

Dynamic matching benefited from all optimizations performed by the pattern-matching compiler, including factorization of tests between cases, and utilization of typing informations. As a consequence, dynamic matching was performed quite efficiently. However, we agreed that this efficiency was not worth the extra complication of the compiler, and this prototype was never carried to the point it could be released.

4 Assessment

This section discusses the practical usefulness of the two propositions above, drawing from our practical experience with the CAML system.

4.1 Interfacing with system functions

Dynamics makes it possible to provide an interface with a number of system functions that cannot be given a static type in ML. Without dynamics, these functions

could not be made available to the user in a type-safe way. In the CAML system, these functions include:

- `extern` : `extern_channel` × `dyn` → `unit` and `intern` : `intern_channel` → `dyn`, to efficiently write and read data structures on persistent storage, preserving sharing inside the structure. A typical use is, for a separate compiler, to communicate compiled object code with its linker, and to save and reload symbol tables representing compiled module interfaces.

- `eval_syntax` : `ML` → `dyn`, to typecheck, compile, and evaluate a piece of abstract ML syntax (type `ML`). This makes it easy to provide CAML as an embedded language inside a program. For instance, the Calculus of Construction [8], a proof development environment, provides the ability to interactively define proof tactics written in CAML, and to apply them on the fly. The CAML macro facility [18, chapter 18] also makes use of `eval_syntax`, since a macro body is an arbitrary CAML expression whose evaluation leads to the substituted text.

- `MLquote` : `dyn` → `ML`, which is one of the constructors of the datatype representing abstract syntax trees. This constructor embeds constants of arbitrary types inside syntax trees. These constants are produced by compile-time evaluation (e.g. macro expansion and constant folding).

- `print` : `dyn` → `unit`, to print a dynamic value in ML syntax. CAML cannot provide a polymorphic printing function with type $\alpha \rightarrow$ `unit`, due to some optimizations in the data representation algorithm, that makes it impossible to decipher the representation of a data without knowing its type.

In these examples, the returned dynamics are generally coerced to fully known types, usually monomorphic. Therefore, we do not see the need for existential type variables there, and the simpler dynamic system presented in section 2 seems largely sufficient. In practice, the restriction encountered first is not that dynamics can only be coerced to closed types, but that dynamics can only be created with closed types. This prevents the `print` function from being called by a polymorphic function to print its polymorphic argument, for instance. This is often needed for debugging purposes.

4.2 Ad-hoc polymorphism

ML polymorphism is uniform: polymorphic functions operate in the same way on arguments of several types. In contrast, ad-hoc polymorphism consists in having generic functions that accept arguments of several types, but operate differently on objects of different types. Prime examples are the `print` function or the `equal` predicate: different algorithms are used to print or compare integers, strings, lists, or references. Several extensions of functional languages have been proposed, that support the definition of such generic functions, including type classes [16] and run-time overloading [15].

Dynamics provide a naive, but easy to understand, way to define generic functions. As demonstrated above in the `print` example, dynamics permit joining predefined functions on atomic types (`print_int`, `print_string`) and functions on data

structures (pairs, lists), that recurse on the components of the structures — the main operation in defining generic functions. Another important aspect of generic functions is extensibility: whenever a new datatype is defined, they can be extended to deal with objects of the new type as well. This can also be supported in the dynamic implementation, by keeping in a reference a list of, say, printing functions with type `dyn → unit`, to be applied until one succeeds whenever none of the standard cases apply.

```
exception Cannot_print;;
type fun_arg = Arg of string in
let printers = ref ([] : (dyn → unit) list) in
let rec print = function
    ...
  | d →
      let rec try_print = function
        [] → print_string "?"
      | f :: rest → try f d with Cannot_print → try_print rest
      in try_print !printers
and new_printer f =
  printers := f :: !printers
```

Assuming, for instance, that type `foo = A | B of int × foo` has been defined, we could add a printer for type `foo` as follows:

```
new_printer (function
    dynamic(A : foo) → print_string "A"
  | dynamic(B(x, y) : foo) →
      print_string "B("; print (dynamic x); print_string ",";
      print (dynamic y); print_string ")"
  | x → raise Cannot_print)
```

It should be pointed out that this implementation of generic functions with dynamics has several major drawbacks. First, because of the restrictions on dynamic creation, polymorphic functions that need to call `print` have to take dynamics themselves. This is not too serious for `print`, but would be prohibitive for heavily used generic functions, such as `equal`: all functions on sets, association lists, etc., would have to operate on dynamics, thus dramatically reducing accuracy of static typing and efficiency of compiled code. Moreover, nothing statically prevents the `print` function from being applied to objects that have no printing method defined. This important class of type errors will only be detected at run-time. Finally, such an implementation of generic functions is rather inefficient, since dynamics are built and coerced at each recursive call.

Type classes and run-time overloading techniques seem more realistic in this respect. They statically guarantee that generic functions can only be applied to objects on which they are defined. They perform type matching at compile-time whenever possible. And run-time type information can usually be arranged as dictionaries of methods, allowing faster method selection than dynamic type matching.

5 Conclusions

We have presented two extensions of ML with dynamic objects. The simpler one has proved quite successful for interfacing user code with some important system functions in a type-safe way. Its implementation cost remains moderate. The other extension, that generalizes the dynamic patterns to include both universal and existential variables in the type part, makes it possible to work on dynamics without coercing them to fixed types. Its semantics are more delicate, and therefore harder to implement. We lack strong evidence of its practical usefulness. We have presented one promising application: writing generic functions such as `print` in a way that is conceptually simpler than type classes. However, the usability of these functions is limited by the restriction that dynamics must be created with closed types. This restriction can be lifted, either by passing type information at run-time to all polymorphic functions, or by examining the call chain at dynamic creation time to reconstruct the instantiations of type variable — a technique developed for tagless garbage collection [3, 6]. It remains to estimate the run-time penalty incurred.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *16th symposium Principles of Programming Languages*. ACM Press, 1989.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. Research report 47, DEC Systems Research Center, 1989. Extended version of [1].

[3] Andrew W. Appel. Run-time tags aren't necessary. *Lisp and Symbolic Computation*, 2(2), 1989.

[4] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer, 1986.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[6] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Programming Language Design and Implementation*, 1991.

[7] Michael Gordon. Adding `eval` to ML. Privately circulated note, circa 1980.

[8] Gérard Huet. The Calculus of Constructions, documentation and user's guide. Technical report 110, INRIA, 1989.

[9] David Maier and David S. Warren. *Computing with logic: logic programming with Prolog*. Benjamin/Cummings, 1988.

[10] Dale Miller. Lexical scoping as universal quantification. In *Proceedings of the sixth international conference for logic programming*, 1989.

[11] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[13] Alan Mycroft. Dynamic types in ML. Draft, 1983.

[14] Simon L. Peyton-Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.

[15] François Rouaix. Safe run-time overloading. In *17th symposium Principles of Programming Languages*. ACM Press, 1990.

[16] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *16th symposium Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[17] David H.D. Warren. An abstract Prolog instruction set. Technical note 309, SRI International, 1983.

[18] Pierre Weis et al. The CAML reference manual, version 2.6.1. Technical report 121, INRIA, 1990.