Journal of Functional Programming, volume 3 number 4, 1993.

Dynamics in ML

Xavier Leroy
École 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 article presents two extensions of the ML language with dynamics, based on our work on 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 the 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` 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, as introduced by

Cardelli [5], is an elegant answer to this need. A dynamic is a pair of a value v and a type expression τ, such that v has type τ. From the standpoint of static typing, all dynamics belong to the built-in type `dyn`. The 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 can be performed.

Continuing the examples above, the 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 [19]. 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 [5, 7]. Abadi et al. [2] mention some of the problems involved with polymorphism, but briefly and informally. They reference a draft paper by Mycroft [21] that is said to consider the extension of ML with dynamics; this article has never been published, and we could not get a copy of it. Recently, Abadi et al. have proposed an extension to their earlier work that allows polymorphism and subtyping [3]. We compare their proposals to ours in section 5 below.

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 [29], the ML implementation developed at INRIA, for more than three 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 significant subset of ML, show the soundness of typing with respect to evaluation, 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 mention some related work in section 5, and give concluding remarks in section 6.

2 Simple dynamics

This section describes dynamics as they are implemented in CAML release 2.6 and later [29, ch. 8].

2.1 Introduction and elimination constructs

The new construct `dynamic` a is provided to create dynamics. This construct evaluates a, and pairs it with (the representation of) the 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.

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 traditionally presented as a special syntactic construct, such as the `typecase` construct in [2]. 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 mechanisms, binding and failure handling, are already provided by the pattern-matching machinery. Hence, instead of providing a separate coercion construct, we integrate dynamic coercion within pattern-matching. 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[1], here is a function that takes a dynamic and attempts to print 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 "?"
```

2.2 Creation of polymorphic dynamics

The introduction of dynamics in a polymorphic type system raises some issues that do not appear in the case of a monomorphic type system. In this section, we show that some restrictions must be put on dynamic creation; the next section deals with the semantics of type matching during dynamic coercion.

It is allowed to create a dynamic of an object with a polymorphic type, provided the type is *closed*: none of the type variables free in the type should be free in the current typing environment. For instance, `dynamic(function x → x)` is perfectly legal, since the identity function has type $\alpha \to \alpha$, and α is a fresh type variable that does not appear anywhere else (in the case of a principal typing). It will be possible to use the internal value of the dynamic with all type instances of $\alpha \to \alpha$.

On the other hand, `function x →` `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 impossible 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

[1] All examples in this article are written in the CAML dialect of ML [8, 16].

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 propagate these types to the polymorphic functions they call.

Besides complicating compilation and raising efficiency issues, passing type information at runtime is essentially incompatible with the ML semantics, because it implies "by-name" semantics for polymorphism instead of ML's "by-value" semantics for polymorphism, in the terminology of [13]. In other terms, the extra abstractions that must be inserted at generalization points to pass type information around cause the evaluation of polymorphic expressions to be delayed until instantiation-time, and therefore side-effects can occur at different times than in ML. Assume for instance that the following expression is not rejected:

```
let f = function x → (print"Hi!"; function y → dynamic(x, y)) in
let g = f(1) in
  (g(1), g(true))
```

To correctly propagate type information, the compiler must insert extra abstractions over type representations and the corresponding applications, as follows:

```
let f = function τ′ → function τ″ → function x →
            (print"Hi!"; function y → dynamic(x, y : τ′ × τ″)) in
let g = function τ → f(int)(τ)(1) in
  (g(int)(1), g(bool)(true))
```

No other placement of abstractions and applications is possible, since abstractions and applications must correspond to the points where generalization and instantiation take place. Hence, the program above prints "`Hi!`" twice, instead of once as in core ML.

Besides this change in semantics for the `let` construct, allowing dynamics to be created with non-closed types also destroys the parametricity properties of the ML type system. It is well-known that, since polymorphic functions can only operate uniformly over objects whose type is a type variable, all functions with a given polymorphic type obey some algebraic laws that depend only on the type [26, 15]. For instance, all functions f with type $\forall\alpha.\,\alpha$ `list` $\rightarrow \alpha$ `list` are such that, for all functions g and lists l,

$$\texttt{map}\ g\ (f\ l) = f(\texttt{map}\ g\ l).$$

This captures the fact that a function f with the type above can only reorder, duplicate and remove some elements from the given list, regardless of the actual value of the elements; hence we get the same results if we apply an arbitrary transformation g to each element of the list either before applying f or after. This is no longer true if we add dynamics without typing restrictions: the function `f` defined by

```
let f = function l →
  match (dynamic l) with
    dynamic(m : int list) → reverse l | d → l
```

would have type $\forall\alpha.\,\alpha$ `list` $\rightarrow \alpha$ `list`, yet

```
map string_of_int (f [1;2]) = ["2";"1"]
f(map string_of_int [1;2])  = ["1";"2"]
```

The closedness condition on dynamic creation rules out the `f` function above. More generally, it is conjectured that the closedness condition suffices to guarantee that similar parametricity results hold for ML plus dynamic and for core ML.

2.3 Coercion of polymorphic dynamics

For type matching in the presence of polymorphic types, two behaviors can be considered. The first one is to require that the internal type of the dynamic be 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 this semantics, identifier `f` can safely be used with different instances of the type $\alpha \to \alpha$ in the right-hand side of the pattern-matching, as in:

```
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 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)` matches only 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 fewer 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`.

2.4 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:

Type expressions:

τ	::=	`int`	a base type
	‖	α	type variable
	‖	$\tau_1 \to \tau_2$	function type
	‖	$\tau_1 \times \tau_2$	product type
	‖	`dyn`	the type of dynamics

Patterns:

p	::=	x	pattern variable
	‖	i	constant pattern
	‖	(p_1, p_2)	pair pattern
	‖	`dynamic`$(p_1 : \tau)$	dynamic pattern

Expressions:

a	::=	x	variable
	‖	i	integer constant
	‖	`function` $p_1 \to a_1 \mid \ldots \mid p_n \to a_n$	function (with pattern matching)
	‖	$a_1\ a_2$	function application
	‖	(a_1, a_2)	pair construction
	‖	`let` $x = a_1$ `in` a_2	the `let` binding
	‖	`dynamic` a_1	dynamic construction

To precisely define the semantics of `dynamic` expressions, we need to keep track of the type given to their arguments during typechecking. For this purpose, we introduce annotated expressions (typical element b), that have the same syntax as raw expressions a, except that the dynamic expressions also contain the type of their argument. More precisely, dynamic expressions are annotated by a type scheme: a type expression with some variables universally quantified.

Type schemes:

σ	::=	$\forall \alpha_1 \ldots \alpha_n . \tau$

Annotated expressions:

b	::=	x	
	‖	`function` $p_1 \to b_1 \mid \ldots \mid p_n \to b_n$	
	‖	`dynamic` $(b_1 : \sigma)$	
	‖	$\ldots$	

Type schemes are identified up to a permutation or renaming of bound variables. Trivial type schemes $\forall . \tau$ are written τ, and identified with type expressions.

2.5 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 function definitions. Two additional rules present the creation and coercion of dynamics.

The rules define the predicate $E \vdash a : \tau \Rightarrow b$, meaning "expression a has type τ in the typing environment E". The b component can be viewed as the typed completion of a (the input expression): b is an annotated expression, with the same structure as a, that records the types

$$(1)\quad E \vdash i : \texttt{int} \Rightarrow i \qquad\qquad (2)\quad \frac{\tau \leq E(x)}{E \vdash x : \tau \Rightarrow x}$$

$$(3)\quad \frac{\vdash p_k : \tau' \Rightarrow E_k \qquad E + E_k \vdash a_k : \tau'' \Rightarrow b_k \quad (k = 1, \ldots, n)}{E \vdash \texttt{function}\ \ldots \mid p_k \rightarrow a_k \mid \ldots\ :\ \tau' \rightarrow \tau''\ \Rightarrow\ \texttt{function}\ \ldots \mid p_k \rightarrow b_k \mid \ldots}$$

$$(4)\quad \frac{E \vdash a_1 : \tau' \rightarrow \tau \Rightarrow b_1 \qquad E \vdash a_2 : \tau' \Rightarrow b_2}{E \vdash a_1\ a_2 : \tau \Rightarrow b_1\ b_2} \qquad (5)\quad \frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad E \vdash a_2 : \tau_2 \Rightarrow b_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \Rightarrow (b_1, b_2)}$$

$$(6)\quad \frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad E + [x \mapsto Clos(\tau_1, FV(E))] \vdash a_2 : \tau \Rightarrow b_2}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau \Rightarrow \texttt{let}\ \ x = b_1\ \ \texttt{in}\ \ b_2}$$

$$(7)\quad \frac{E \vdash a : \tau \Rightarrow b \qquad FV(\tau) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ \ a : \texttt{dyn} \Rightarrow \texttt{dynamic}(b, Clos(\tau, \emptyset))}$$

$$(8)\quad \vdash x : \tau \Rightarrow [x \mapsto \tau] \qquad\qquad (9)\quad \vdash i : \texttt{int} \Rightarrow [\,]$$

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

Figure 1: Typing rules

given to the arguments of dynamic expressions. The rules make use of an auxiliary predicate, $\vdash p : \tau \Rightarrow E$, meaning "pattern p has type τ and enriches the type environment by E". Here, E stands for a finite mapping from variable names to type schemes.

Some notations on maps. The empty map is written $[\,]$. The map that associates σ to x and is undefined on other variables is written $[x \mapsto \sigma]$. The asymmetric join of two maps E_1 and E_2 is written $E_1 + E_2$; it is asymmetric in the sense that if x belongs to the domain of E_1 and to the domain of E_2, we take $(E_1 + E_2)(x) = E_2(x)$ and ignore $E_1(x)$. The symmetric join of E_1 and E_2 is written $E_1 \oplus E_2$; it is undefined if the domains of E_1 and E_2 are not disjoint.

We write $FV(\tau)$ for the set of all type variables that appear in the type expression τ. For type schemes, we take $FV(\sigma)$ to be the type variables free in σ: $FV(\forall\alpha_1 \ldots \alpha_n.\, \tau) = FV(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}$. Similarly, $FV(E)$ is the union of the free variables of all type schemes in the codomain of E.

We write $\tau \leq \sigma$ to express that type τ is an instance of type scheme σ; that is, writing $\sigma = \forall\alpha_1 \ldots \alpha_n.\, \tau'$, we have $\tau = \tau'\{\alpha_1 \leftarrow \tau_1 \ldots \alpha_n \leftarrow \tau_n\}$ for some types $\tau_1 \ldots \tau_n$.

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 rules that significantly differ from those of the core ML language are rule 7, that deals with dynamic creation, and rule 11, that deals with dynamic coercion. Rule 7 says that the expression $\texttt{dynamic}(a)$ has type $\texttt{dyn}$, provided that a has a type τ that can be closed: none of the free variables of τ are free in the current typing environment E. The completion of $\texttt{dynamic}\ a$ is $\texttt{dynamic}(b : \sigma)$, where b is the completion of a, and σ the type scheme obtained by generalizing all variables free in τ, that is, $Clos(\tau, \emptyset)$.

Rule 11 says that the pattern $\texttt{dynamic}(p : \tau)$ matches values of type $\texttt{dyn}$, provided that 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, $\texttt{dynamic}(p : \tau)$ 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 $\texttt{dynamic}(p : \tau)$ binds $x_1 \ldots x_n$ to values of types $Clos(\tau_1, \emptyset) \ldots Clos(\tau_n, \emptyset)$.

One might be surprised by the fact that the types $\tau_1 \ldots \tau_n$ are generalized independently. For instance, the pattern $\texttt{dynamic}(\texttt{f}, \texttt{g} : (\alpha \to \alpha) \times (\alpha \to \alpha))$ results in the environment

$$[\texttt{f} : \forall\alpha.\, \alpha \to \alpha,\ \texttt{g} : \forall\beta.\, \beta \to \beta]$$

(after renaming α to β in the second type scheme), and the information that the types of $\texttt{f}$ and $\texttt{g}$ share the same α has been lost. Actually, this makes no difference, because the type schemes $\forall\alpha.\, (\alpha \to \alpha) \times (\alpha \to \alpha)$ and $\forall\alpha, \beta.\, (\alpha \to \alpha) \times (\beta \to \beta)$ are isomorphic: all terms belonging to one type scheme also belong to the other [10]. The isomorphisms between polymorphic product types justify the independent generalization of the components of E in rule 11.

An important property of the ML type system is the stability of typing judgements under substitutions [9]. The typing predicate defined above also enjoys this property.

Proposition 1 *If $E \vdash a : \tau \Rightarrow b$, then $\varphi(E) \vdash a : \varphi(\tau) \Rightarrow b$ for all substitutions φ.*

Proof: First, we check that if $\vdash p : \tau \Rightarrow E$, then $\vdash p : \varphi(\tau) \Rightarrow \varphi(E)$. This is an easy inductive argument on p. The case where p is a dynamic pattern follows from the fact that $\varphi(Clos(E, \emptyset)) = Clos(E, \emptyset)$, since $Clos(E, \emptyset)$ has no free variables. Then, proposition 1 follows from a well-known

inductive argument on a [20, section 4.7.2]. We give only the new case: $a = \texttt{dynamic}\ a_1$. In this case, the typing derivation is:

$$\frac{E \vdash a : \tau \Rightarrow b \qquad FV(\tau) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ a : \texttt{dyn} \Rightarrow \texttt{dynamic}(b : Clos(\tau, \emptyset))}$$

By renaming if necessary, we can assume that none of the free variables of τ are (1) in the domain of φ, and (2) free in $\varphi(E)$. This renaming does not modify E, since $FV(\tau) \cap FV(E) = \emptyset$. Applying the induction hypothesis, we get a proof of $\varphi(E) \vdash a : \varphi(\tau) \Rightarrow b$. We have $\varphi(\tau) = \tau$ by hypothesis (1), and $FV(\tau) \cap FV(\varphi(E)) = \emptyset$ by hypothesis (2). Hence we can conclude $\varphi(E) \vdash \texttt{dynamic}\ a : \texttt{dyn} \Rightarrow \texttt{dynamic}(b : Clos(\tau, \emptyset))$, which is the expected result. □

2.6 Evaluation

We now give call-by-value operational semantics for our calculus. Annotated expressions b are mapped to responses (ranged over by r). Responses are either values, or the constant `wrong` that denotes run-time type violations. Values (v) are terms with the following syntax:

Values:

v	::=	$\texttt{cst}(i)$	integer value
	∥	$\texttt{pair}(v, v')$	value pair
	∥	$\texttt{dynamic}(v : \sigma)$	dynamic value
	∥	$\texttt{clos}(e,\ p_1 \to b_1 \mid \ldots \mid p_n \to b_n)$	function closure

Evaluation environments:

e	::=	$[\ldots, x \mapsto v, \ldots]$	

Evaluation responses:

r	::=	v	normal response (a value)
	∥	`wrong`	type error response

Pattern-matching responses:

m	::=	e	normal response (an environment)
	∥	`wrong`	type error response

The type schemes in dynamic values are required to be closed: all variables are universally quantified.

The evaluation rules are given in figure 2. They closely follow the structure of the typing rules. The first two sets of rules define the predicate $e \vdash b \xrightarrow{*} r$, meaning "in environment e, expression b evaluates to response r". The last two sets of rules define the auxiliary predicate $\vdash v < p \xrightarrow{*} m$, meaning "the matching of value v against pattern p results in m". Here, m is either an evaluation environment, describing the bindings performed on p variables in case of successful matching, or the constant `wrong` if a run-time type violation occurred.

Since most rules are similar to those of ML [19], we only detail the two rules dealing with dynamics. Rule 18 expresses that evaluating $\texttt{dynamic}(b : \sigma)$ amounts to evaluating b and pairing its value with σ, the static type of b. Rule 29 defines the semantics of pattern matching over dynamics. The internal type scheme σ of the dynamic is required to be more general than the type τ' expected by the pattern: the type τ' must be an instance of the type scheme σ, in the sense

$$(12)\quad e \vdash x \xrightarrow{*} e(x) \qquad (13)\quad e \vdash i \xrightarrow{*} \texttt{cst}(i)$$

$$(14)\quad e \vdash \texttt{function}\ \ldots \mid p_k \to b_k \mid \ldots \xrightarrow{*} \texttt{clos}(e, \ldots \mid p_k \to b_k \mid \ldots)$$

$$(15)\quad \frac{\begin{array}{c} e \vdash b' \xrightarrow{*} \texttt{clos}(e', \ldots \mid p_k \to b_k \mid \ldots) \qquad e \vdash b'' \xrightarrow{*} v \\ \vdash v < p_k \xrightarrow{*} e'' \qquad e' + e'' \vdash b_k \xrightarrow{*} r \qquad k \text{ minimal} \end{array}}{e \vdash b'\ b'' \xrightarrow{*} r}$$

$$(16)\quad \frac{e \vdash b_1 \xrightarrow{*} v_1 \qquad e \vdash b_2 \xrightarrow{*} v_2}{e \vdash (b_1, b_2) \xrightarrow{*} \texttt{pair}(v_1, v_2)} \qquad (17)\quad \frac{e \vdash b_1 \xrightarrow{*} v \qquad e + [x \mapsto v] \vdash b_2 \xrightarrow{*} r}{e \vdash \texttt{let}\ x = b_1\ \texttt{in}\ b_2 \xrightarrow{*} r}$$

$$(18)\quad \frac{e \vdash b \xrightarrow{*} v}{e \vdash \texttt{dynamic}(b : \sigma) \xrightarrow{*} \texttt{dynamic}(v : \sigma)}$$

$$(19)\quad \frac{e \vdash b' \xrightarrow{*} v \qquad v \neq \texttt{clos}(e, \ldots)}{e \vdash b'\ b'' \xrightarrow{*} \texttt{wrong}} \qquad (20)\quad \frac{e \vdash b'' \xrightarrow{*} \texttt{wrong}}{e \vdash b'\ b'' \xrightarrow{*} \texttt{wrong}}$$

$$(21)\quad \frac{e \vdash b' \xrightarrow{*} \texttt{clos}(e_1, p_1 \to b_1 \mid \ldots \mid p_n \to b_n) \qquad e \vdash b'' \xrightarrow{*} v \qquad \vdash v < p_k \xrightarrow{*} \texttt{wrong}}{e \vdash b'\ b'' \xrightarrow{*} \texttt{wrong}}$$

$$(22)\quad \frac{e \vdash b_1 \xrightarrow{*} \texttt{wrong}}{e \vdash (b_1, b_2) \xrightarrow{*} \texttt{wrong}} \qquad (23)\quad \frac{e \vdash b_2 \xrightarrow{*} \texttt{wrong}}{e \vdash (b_1, b_2) \xrightarrow{*} \texttt{wrong}}$$

$$(24)\quad \frac{e \vdash b_1 \xrightarrow{*} \texttt{wrong}}{e \vdash \texttt{let}\ x = b_1\ \texttt{in}\ b_2 \xrightarrow{*} \texttt{wrong}} \qquad (25)\quad \frac{e \vdash b \xrightarrow{*} \texttt{wrong}}{e \vdash \texttt{dynamic}\ (b : \sigma) \xrightarrow{*} \texttt{wrong}}$$

$$(26)\quad \vdash v < x \xrightarrow{*} [x \mapsto v] \qquad (27)\quad \vdash \texttt{cst}(i) < i \xrightarrow{*} [\,]$$

$$(28)\quad \frac{\vdash v_1 < p_1 \xrightarrow{*} e_1 \qquad \vdash v_2 < p_2 \xrightarrow{*} e_2}{\vdash \texttt{pair}(v_1, v_2) < (p_1, p_2) \xrightarrow{*} e_1 \oplus e_2} \qquad (29)\quad \frac{\vdash v < p \xrightarrow{*} e \qquad \tau \leq \sigma}{\vdash \texttt{dynamic}(v : \sigma) < \texttt{dynamic}(p : \tau) \xrightarrow{*} e}$$

$$(30)\quad \frac{v \neq \texttt{cst}(i')}{\vdash v < i \xrightarrow{*} \texttt{wrong}} \qquad (31)\quad \frac{v \neq \texttt{pair}(v_1, v_2)}{\vdash v < (p_1, p_2) \xrightarrow{*} \texttt{wrong}}$$

$$(32)\quad \frac{\vdash v_1 < p_1 \xrightarrow{*} \texttt{wrong}}{\vdash \texttt{pair}(v_1, v_2) < (p_1, p_2) \xrightarrow{*} \texttt{wrong}} \qquad (33)\quad \frac{\vdash v_2 < p_2 \xrightarrow{*} \texttt{wrong}}{\vdash \texttt{pair}(v_1, v_2) < (p_1, p_2) \xrightarrow{*} \texttt{wrong}}$$

$$(34)\quad \frac{v \neq \texttt{dynamic}(v' : \sigma)}{\vdash v < \texttt{dynamic}(p : \tau) \xrightarrow{*} \texttt{wrong}} \qquad (35)\quad \frac{\vdash \tau \leq \sigma \qquad v < p \xrightarrow{*} \texttt{wrong}}{\vdash \texttt{dynamic}(v : \sigma) < \texttt{dynamic}(p : \tau) \xrightarrow{*} \texttt{wrong}}$$

Figure 2: Evaluation rules

of the $\leq$ relation used in the typing rules. Then, the internal value of the dynamic is recursively matched against the value part of the dynamic pattern.

2.7 Soundness

In this section, we show that the typing rules are sound with respect to the evaluation rules: the evaluation of a well-typed program never stops because of a run-time type error, such as trying to apply an integer as if it were a function. In this situation, the evaluation rules given above associate `wrong` to the program. We now show that this cannot occur to a well-typed program.

Proposition 2 *Let a_0 be a program (a closed expression). If $[\,] \vdash a_0 : \tau_0 \Rightarrow b_0$ for some type τ_0, then we cannot derive $[\,] \vdash b_0 \xrightarrow{*} \texttt{wrong}$.*

To prove this result, we first define a semantic typing relation, $\models v : \tau$, saying whether the value v semantically belongs to the type τ. The relation is defined by structural induction on v, as follows.

- $\models \texttt{cst}(i) : \tau$ if and only if $\tau = \texttt{int}$
- $\models \texttt{pair}(v_1, v_2) : \tau$ if and only if $\tau = \tau_1 \times \tau_2$ and $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$
- $\models \texttt{clos}(e,\ p_1 \to b_1 \mid \ldots \mid p_n \to b_n) : \tau$ if and only if $\tau = \tau_1 \to \tau_2$ and there exists a typing environment E and raw expressions $a_1 \ldots a_n$ such that $\models e : E$ and $E \vdash (\texttt{function}\ p_1 \to a_1 \mid \ldots \mid p_n \to a_n) : \tau_1 \to \tau_2 \Rightarrow (\texttt{function}\ p_1 \to b_1 \mid \ldots \mid p_n \to b_n)$
- $\models \texttt{dynamic}(v : \sigma) : \tau$ if and only if $\tau = \texttt{dyn}$ and $\models v : \sigma$.

The use of the typing relation to define $\models$ over functional values is taken from [25]. The semantic typing relation extends to type schemes and to environments:

- $\models v : \sigma$ if and only if $\models v : \tau$ for all types $\tau \leq \sigma$
- $\models e : E$ if and only if e and E have the same domain, and $\models e(x) : E(x)$ for all x in their domain.

The following property shows that semantic typing is stable under substitution; hence, type generalization is always semantically correct.

Proposition 3 *Assume $\models v : \tau$. Then, $\models v : \varphi(\tau)$ for all substitutions φ. Hence $\models v : \forall\alpha_1 \ldots \alpha_n.\tau$ for all variables $\alpha_1 \ldots \alpha_n$.*

Proof: By induction on v. The case where v is a closure is settled by proposition 1. The remaining cases are obvious. □

The two claims below are the inductive steps that establish proposition 2.

Proposition 4 *Assume $\vdash p : \tau \Rightarrow E$ and $\vdash v < p \xrightarrow{*} m$. If $\models v : \tau$, then $m \neq \texttt{wrong}$; instead, m is an evaluation environment e such that $\models e : E$.*

Proof: By induction over p. The cases $p = x$ and $p = i$ are obvious. If $p = (p_1, p_2)$, then the last rule applied in the typing derivation is

$$\frac{\vdash p_1 : \tau_1 \Rightarrow E_1 \qquad \vdash p_2 : \tau_2 \Rightarrow E_2}{\vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow E_1 \oplus E_2}$$

with $\tau = \tau_1 \times \tau_2$ and $E = E_1 \oplus E_2$. By hypothesis $\models v : \tau_1 \times \tau_2$, we have $v = \texttt{pair}(v_1, v_2)$ with $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$. The last rule used in the evaluation derivation is one of 28, 31, 32 or 33. Rule 31 does not apply, since v is a pair. Rule 32 requires $\vdash v_1 < p_1 \Rightarrow \texttt{wrong}$, but this contradicts the induction hypothesis applied to v_1. Rule 33 is similarly excluded. Hence the last evaluation rule used must be

$$\frac{\vdash v_1 < p_1 \xrightarrow{*} e_1 \qquad \vdash v_2 < p_2 \xrightarrow{*} e_2}{\vdash \texttt{pair}(v_1, v_2) < (p_1, p_2) \xrightarrow{*} e_1 \oplus e_2}$$

with $e = e_1 \oplus e_2$. Applying the induction hypothesis to the matching of v_1 against p_1 and to the matching of v_2 against p_2, we get $\models e_1 : E_1$ and $\models e_2 : E_2$. Hence $\models e_1 \oplus e_2 : E_1 \oplus E_2$, which is the expected result.

For the case $p = \texttt{dynamic}(p_1 : \tau_1)$, the last rule applied in the typing derivation is

$$\frac{\vdash p_1 : \tau_1 \Rightarrow E_1}{\vdash \texttt{dynamic}(p_1 : \tau_1) : \texttt{dyn} \Rightarrow Clos(E_1, \emptyset)}$$

with $\tau = \texttt{dyn}$ and $E = Clos(E_1, \emptyset)$. By definition of $\models$, we have $v = \texttt{dynamic}(v_1 : \sigma)$ and $\models v_1 : \sigma$. There are three evaluation possibilities. Rule 34 does not apply, since v is a dynamic. Rule 35 assumes $\tau_1 \leq \sigma$ and $v_1 < p \xrightarrow{*} \texttt{wrong}$. This contradicts the induction hypothesis, since $\models v_1 : \sigma$ implies $\models v_1 : \tau_1$ by definition of $\models$ over type schemes. Hence the last evaluation rule used is

$$\frac{\vdash v_1 < p_1 \xrightarrow{*} e \qquad \tau_1 \leq \sigma}{\vdash \texttt{dynamic}(v_1 : \sigma) < \texttt{dynamic}(p_1 : \tau_1) \xrightarrow{*} e}$$

Since $\models v_1 : \sigma$ and $\tau_1 \leq \sigma$, it follows that $\models v_1 : \tau_1$. Applying the induction hypothesis to the matching of $v_1 : \tau_1$ against p_1, we get $\models e : E_1$. By proposition 3, this implies $\models e : Clos(E_1, \emptyset)$. That's the expected result. □

Proposition 5 *Assume $E \vdash a : \tau \Rightarrow b$ and $e \vdash b \xrightarrow{*} r$. If $\models e : E$, then $r \neq \texttt{wrong}$; instead, r is a value v such that $\models v : \tau$.*

Proof: By induction over the length of the evaluation, and case analysis over a. We show one base case and two inductive cases; the other cases are similar. If $a = x$, the rule used in the typing derivation is

$$\frac{\tau \leq E(x)}{E \vdash x : \tau \Rightarrow x}$$

The only possible evaluation is $e \vdash x \xrightarrow{*} e(x)$. Hence, $r = e(x) \neq \texttt{wrong}$. By hypothesis $\models e : E$, we have $\models e(x) : E(x)$. Since $\tau \leq E(x)$, this implies $\models e(x) : \tau$, which is the expected result. If $a = \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2$, the last typing rule used is

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad E + [x \mapsto Clos(\tau_1, FV(E))] \vdash a_2 : \tau \Rightarrow b_2}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau \Rightarrow \texttt{let}\ \ x = b_1\ \ \texttt{in}\ \ b_2}$$

Two evaluation rules lead to the conclusion $e \vdash b \xrightarrow{*} r$. The first one is

$$\frac{e \vdash b_1 \xrightarrow{*} \texttt{wrong}}{e \vdash \texttt{let}\ \ x = b_1\ \ \texttt{in}\ \ b_2 \xrightarrow{*} \texttt{wrong}}$$

By induction hypothesis, we cannot derive $e \vdash b_1 \xrightarrow{*} \texttt{wrong}$, since $E \vdash a_1 : \tau_1 \Rightarrow b_1$. Hence the last evaluation step must be

$$\frac{e \vdash b_1 \xrightarrow{*} v \qquad e + [x \mapsto v] \vdash b_2 \xrightarrow{*} r}{e \vdash \texttt{let}\ \ x = b_1\ \ \texttt{in}\ \ b_2 \xrightarrow{*} r}$$

Applying the induction hypothesis to the evaluation of b_1, we get $\models v : \tau_1$. By proposition 3, this implies $\models v : Clos(\tau_1, FV(E))$. Hence $\models e + [x \mapsto v] : E + [x \mapsto Clos(\tau_1, FV(E))]$, and we can apply the induction hypothesis to the evaluation of b_2. This leads to the expected result: $r \neq \texttt{wrong}$ and $\models r : \tau_2$.

If $a = \texttt{dynamic}\ a_1$, the last typing rule used is

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad FV(\tau_1) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ \ a : \texttt{dyn} \Rightarrow \texttt{dynamic}(b_1, Clos(\tau_1, \emptyset))}$$

There are two evaluation possibilities. The first one (rule 25) concludes $r = \texttt{wrong}$ because $e \vdash b_1 \xrightarrow{*} \texttt{wrong}$; but this contradicts the induction hypothesis. Hence the last evaluation step must be

$$\frac{e \vdash b_1 \xrightarrow{*} v_1}{e \vdash \texttt{dynamic}(b_1 : Clos(\tau_1, \emptyset)) \xrightarrow{*} \texttt{dynamic}(v_1 : Clos(\tau_1, \emptyset))}$$

Applying the induction hypothesis to the evaluation of b_1, we get $\models v_1 : \tau_1$. By proposition 3, this implies $\models v_1 : Clos(\tau_1, \emptyset)$. Hence $\models \texttt{dynamic}(v_1 : Clos(\tau_1, \emptyset)) : \texttt{dyn}$, as expected. □

2.8 Type reconstruction

Unlike the ML type system, the type system presented above does not possess the principal type property. This fact is due to the closedness condition in the rule for dynamic expressions. Consider the expression $\texttt{function}\ x \to \texttt{dynamic}\ x$. It has types $\texttt{int} \to \texttt{dyn}$, and $\texttt{dyn} \to \texttt{dyn}$, and more generally $\tau \to \texttt{dyn}$ for all closed types τ; but the lower bound of all these types, $\alpha \to \texttt{dyn}$, is not a valid type for the function, since it corresponds to the construction of a dynamic with a statically unknown type. Such programs must be statically detected, and rejected as ambiguous: they have no well-defined semantics. The programmer must put more type constraints to unambiguously state what the program should do.

$P(i) = (\texttt{int}, [\,])$

$P(x) = (\alpha, [x \mapsto \alpha])$
where α is a fresh type variable

$P(p_1, p_2) = (\tau_1 \times \tau_2,\ E_1 \oplus E_2)$
where $(\tau_1, E_1) = P(p_1)$ and $(\tau_2, E_2) = P(p_2)$

$P(\texttt{dynamic}(p_1 : \tau)) = (\tau, \theta E_1)$
where $(\tau_1, E_1) = P(p_1)$ and θ is such that $\theta\tau_1 = \tau$

$W(i, E) = (\texttt{int}, id, \emptyset)$

$W(x, E) = (\tau\{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\},\ id,\ \emptyset)$
where $E(x) = \forall\alpha_1 \ldots \alpha_n.\, \tau$ and $\beta_1, \ldots, \beta_n$ are fresh variables

$W((\texttt{function}\ p_1 \to a_1 \mid \ldots \mid p_n \to a_n), E) = (\tau' \to \tau'',\ \varphi,\ D)$
where $(\tau'_k, E_k) = P(p_k)$ for $k = 1, \ldots, n$
and $\mu' = \texttt{mgu}(\tau'_1, \ldots, \tau'_n)$
and $(\tau''_1, \varphi_1, D_1) = W(a_1, E + \mu' E_1)$
and $(\tau''_2, \varphi_2, D_2) = W(a_2, \varphi_1(E + \mu' E_2))$
and $\ldots$
and $(\tau''_n, \varphi_n, D_n) = W(a_n, \varphi_{n-1} \ldots \varphi_1(E + \mu' E_n))$
and $\mu'' = \texttt{mgu}(\varphi_n \ldots \varphi_2 \tau''_1, \ldots, \varphi_n \tau''_{n-1}, \tau''_n)$
and $\varphi = \mu'' \circ \varphi_n \circ \cdots \circ \varphi_1 \circ \mu'$
and $\tau' = \varphi\tau'_1$ and $\tau'' = \mu''\tau''_n$
and $D = \mu''\varphi_n \ldots \varphi_2 D_1 \cup \ldots \cup \mu''\varphi_n D_{n-1} \cup \mu'' D_n$

$W(a_1(a_2), E) = (\mu\alpha,\ \mu \circ \varphi_2 \circ \varphi_1,\ \mu\varphi_2 D_1 \cup \mu D_2)$
where $(\tau_1, \varphi_1, D_1) = W(a_1, E)$
and $(\tau_2, \varphi_2, D_2) = W(a_2, \varphi_1 E)$
and $\mu = \texttt{mgu}(\varphi_2\tau_1, \tau_2 \to \alpha)$ with α being a fresh variable

$W(\texttt{let}\ x = a_1\ \texttt{in}\ a_2, E) = (\tau_2,\ \varphi_2 \circ \varphi_1,\ \varphi_1(D_1) \cup D_2)$
where $(\tau_1, \varphi_1, D_1) = W(a_1, E)$
and $(\tau_2, \varphi_2, D_2) = W(a_2, \varphi_1 E + x \mapsto \forall\alpha_1 \ldots \alpha_n.\, \tau_1)$
with $\{\alpha_1 \ldots \alpha_n\} = FV(\tau_1) \setminus FV(\varphi_1 E) \setminus FV(D_1)$

$W(\texttt{dynamic}(a_1), E) = (\texttt{dyn}, \varphi_1, D_1 \cup (FV(\tau_1) \cap FV(\varphi_1 E)))$
where $(\tau_1, \varphi_1, D_1) = W(a_1, E)$

$W((a_1, a_2), E) = (\varphi_2\tau_1 \times \tau_2,\ \varphi_2 \circ \varphi_1,\ \varphi_2 D_1 \cup D_2)$
where $(\tau_1, \varphi_1, D_1) = W(a_1, E)$
and $(\tau_2, \varphi_2, D_2) = W(a_2, \varphi_1 E)$

Figure 3: The type reconstruction algorithm

There is a slight technical difficulty in detecting ambiguous programs. For the `dynamic` a construct, 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 later be instantiated to closed types. 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 is instantiated to `int`, and we know that the dynamic is created with internal type `int`. Hence, the closedness check must be delayed until the end of type inference. The idea is as follows: when typing `dynamic` a, we record all type variables that are free in the inferred type for a and in the current typing environment. At the end of typechecking, all type variables in this set must be instantiated to closed types.

This process can be formalized as a simple extension of the Damas-Milner algorithm W [9]. The algorithm is shown in figure 3. It takes as input an expression a and an initial typing environment E. It outputs a type τ (the type inferred for a), a substitution φ (recording instantiations performed on E), and a set D of type expressions that keeps track of ambiguous dynamic constructions.[2] Each time a `dynamic`(a) expression is typed, all type variables free in the inferred type for a and in the current typing environment are added to D. For expressions of other kinds, D is simply carried around; the instantiations performed on E are also performed on D. Variables free in D are prevented from being generalized.

If the algorithm fails, the program is not well-typed. If the algorithm succeeds with D containing only closed types, then τ is the most general type for the program. If the algorithm succeeds, but D contains non-closed types, then the program is ambiguous; however, it has type $\psi(\tau)$ for all substitutions ψ such that $\psi(D)$ contains only closed types.

For dynamic patterns `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.

2.9 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. All type variables are assumed universally quantified. The code generated for `dynamic`$(b : \sigma)$ simply pairs the value of a with the structured constant representing a trivial instance of σ 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

[2] Algorithm W should also produce an annotated expression b. This extra result has been omitted for the sake of simplicity. It is easy to reconstruct b from the principal typing derivation built by the algorithm.

gtype). The sequence of tests matching the internal value against the pattern is entered only when ge_gtype returns true. 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.

To speed up run-time type tests, we could 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 [6]. These physical modifications are reversed at the end of matching.

To make dynamic coercions even faster, 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 [22] 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 is actually just as difficult 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 [28, 14, 4]. 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((x, 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

We first show how existentially quantified type variables behave when typing the right-hand side of the pattern-matching. An existentially quantified variable can be bound to any actual type at run-time. Hence, at compile-time, we should make no assumptions about the type α, and treat it as an abstract type. That is, the type α does not match any type except itself; and 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 illustrate 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, the variable α matches only type expressions that do not depend on β. For instance, a dynamic with internal type $\forall\gamma.\,\gamma \to \gamma$ 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 normally, 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 to type expressions containing α. For instance, this pattern matches a dynamic with type $\forall\gamma.\,\gamma \to \gamma$ `list`, binding β to α `list`. This pattern catches a class of functions that operate uniformly on arguments of any type [26]. These functions cannot test or destructure their arguments, but only put them 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.[3] (More exactly, 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 considering β in the right-hand side of the matching as a type constructor parameterized by α. (This transformation is known in logic as Skolemization.) To avoid confusion, we shall write S_β for the type constructor associated to the type variable β. Therefore, we now

[3] 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.

assume $\texttt{f} : \forall\alpha.\ \alpha \to S_\beta(\alpha)$ for the typing of $\texttt{f(1)} = \texttt{f(true)}$, and this leads to a static type error, since the two sides of the equal sign have incompatible types $S_\beta(\texttt{int})$ and $S_\beta(\texttt{bool})$. However, $\texttt{f(1)} = \texttt{f(2)}$ is well-typed, since both sides have type $S_\beta(\texttt{int})$. The general rule is: for the purpose of typing the right-hand side of a pattern-matching, an existentially quantified type variable β is replaced by the type expression $S_\beta(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1 \ldots \alpha_n$ is the list of those type variables that are universally quantified before β in the pattern.

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 (because of the type isomorphisms). 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 [2] 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 as it appears in [2]:

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

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 lead to ambiguities in the bindings of types to existentially quantified type variables. Consider:

$$\begin{array}{l}\texttt{let f} = \texttt{function}\ \exists\alpha.\,\texttt{dynamic}(\texttt{x} : \alpha) \to \\ \qquad\qquad \texttt{let d} = \texttt{dynamic}(\texttt{x})\ \texttt{in function dynamic}(\texttt{y} : \alpha) \to \texttt{d} \\ \texttt{in f}(\texttt{dynamic}\ [\,])(\texttt{dynamic}\ [1])\end{array}$$

The first type matching succeeds with α bound to $\forall\beta.\,\beta\ \texttt{list}$, but the second matching requires α to be narrowed to $\texttt{int list}$. It is unclear whether the dynamic $\texttt{d}$ created between the two matchings should have internal type $\forall\beta.\,\beta\ \texttt{list}$ or $\texttt{int list}$. This problem does not arise if we require the actual type bound to α to be determined by only one matching. This is ensured by the closedness condition on pattern matching, 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 cost of a later error detection, if $\texttt{df}$ is not a dynamic of a function.

$$(36)\quad E \vdash i : \texttt{int} \Rightarrow i \qquad (37)\quad \frac{\tau \leq E(x)}{E \vdash x : \tau \Rightarrow x}$$

$$(38)\quad \frac{Q_k \vdash p_k : \tau' \Rightarrow E_k \qquad E + E_k \vdash a_k : \tau'' \Rightarrow b_k \qquad FSC(\tau'') \cap BV(Q_k) = \emptyset}{E \vdash \texttt{function}\ \ldots \mid Q_k.p_k \to a_k \mid \ldots : \tau' \to \tau'' \Rightarrow \texttt{function}\ \ldots \mid Q_k.p_k \to b_k \mid \ldots}$$

$$(39)\quad \frac{E \vdash a_1 : \tau' \to \tau \Rightarrow b_1 \qquad E \vdash a_2 : \tau' \Rightarrow b_2}{E \vdash a_1\ a_2 : \tau \Rightarrow b_1\ b_2} \qquad (40)\quad \frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad E \vdash a_2 : \tau_2 \Rightarrow b_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \Rightarrow (b_1, b_2)}$$

$$(41)\quad \frac{E \vdash a_1 : \tau_1 \Rightarrow b_1 \qquad E + [x \mapsto Clos(\tau_1, FV(E))] \vdash a_2 : \tau \Rightarrow b_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau \Rightarrow \texttt{let}\ x = b_1\ \texttt{in}\ b_2}$$

$$(42)\quad \frac{E \vdash a : \tau \Rightarrow b \qquad FV(\tau) \cap FV(E) = \emptyset}{E \vdash \texttt{dynamic}\ a : \texttt{dyn} \Rightarrow \texttt{dynamic}(b, Clos(\tau, \emptyset))}$$

$$(43)\quad Q \vdash x : \tau \Rightarrow [x \mapsto \tau] \qquad (44)\quad \frac{Q \vdash p_1 : \tau_1 \Rightarrow E_1 \qquad Q \vdash p_2 : \tau_2 \Rightarrow E_2}{Q \vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow E_1 \oplus E_2}$$

$$(45)\quad Q \vdash i : \texttt{int} \Rightarrow [\,] \qquad (46)\quad \frac{FV(\overline{\tau}) \subseteq BV(Q) \qquad Q \vdash p : \overline{\tau} \Rightarrow E \qquad \theta = S(\epsilon, Q)}{Q \vdash \texttt{dynamic}(p : \overline{\tau}) : \texttt{dyn} \Rightarrow Clos(\theta(E), \emptyset)}$$

Figure 4: Typing rules with mixed quantification in type patterns

3.2 Syntax

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

Expressions:

$a \ ::= \ \ldots \parallel \texttt{function}\ Q_1.p_1 \to a_1 \mid \ldots \mid Q_n.p_n \to a_n$

Annotated expressions:

$b \ ::= \ \ldots \parallel \texttt{function}\ Q_1.p_1 \to b_1 \mid \ldots \mid Q_n.p_n \to b_n$

Quantifier prefixes:

$Q \ ::= \ \epsilon \parallel \forall\alpha.Q \parallel \exists\alpha.Q$

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 S_α, with variable arity.

$$\tau \quad ::= \quad \ldots \;\|\; S_\alpha(\tau_1, \ldots, \tau_n)$$

Skolem constants appear only inside inferred types: they cannot appear in the type part of dynamic patterns, nor in the internal type of dynamic values. We shall write $\overline{\tau}$, $\overline{\sigma}$ for types that do not contain Skolem constants. We define $FSC(\tau)$, the free Skolem constants of type τ, as the set of all variables α such that the type constructor S_α appears in τ.

The new typing rules for functions and for patterns are shown in figure 4. For each case $Q.p \to a$ 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:

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

The typing of the action a proceeds as previously. We simply check that the type of a does not contain any Skolem constants corresponding to variables bound by Q. This is ensured by the side-condition $FSC(\tau'') \cap BV(Q_k) = \emptyset$ in rule 38.

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 5. The value space used is:

Values:

v	::=	$\texttt{cst}(i)$	integer value
	$\|$	$\texttt{pair}(v, v')$	value pair
	$\|$	$\texttt{dynamic}(v : \sigma)$	dynamics value
	$\|$	$\texttt{clos}(e,\ Q_1.p_1 \to b_1 \mid \ldots \mid Q_n.p_n \to b_n)$	function closure

Evaluation environments:

e	::=	$[\ldots, x \mapsto v, \ldots, \alpha \mapsto (\lambda\alpha_1 \ldots \alpha_n.\, \overline{\tau}), \ldots]$	

Responses:

r	::=	v	normal response
	$\|$	$\texttt{wrong}$	type error response

Pattern-matching responses:

m	::=	$(e,\ \Gamma)$	normal response
	$\|$	$\texttt{wrong}$	type error response

The type schemes appearing in dynamic values are required to be closed, and not to contain any Skolem constant.

$$(47)\quad e \vdash x \xrightarrow{*} e(x) \qquad\qquad (48)\quad e \vdash i \xrightarrow{*} \texttt{cst}(i)$$

$$(49)\quad e \vdash \texttt{function}\ \ \ldots \mid Q_k.p_k \to b_k \mid \ldots \xrightarrow{*} \texttt{clos}(e, \ldots \mid Q_k.p_k \to b_k \mid \ldots)$$

$$(50)\quad \frac{e \vdash b' \xrightarrow{*} \texttt{clos}(e', \ldots \mid Q_k.p_k \to b_k \mid \ldots) \qquad e \vdash b'' \xrightarrow{*} v \qquad Q_k \vdash v < p_k \xrightarrow{*} (e'',\ \Gamma) \qquad e' + e'' + Solve(Q_k, \Gamma) \vdash a_k \xrightarrow{*} r \qquad k \text{ minimal}}{e \vdash b'\ b'' \xrightarrow{*} r}$$

$$(51)\quad \frac{e \vdash b_1 \xrightarrow{*} v_1 \qquad e \vdash b_2 \xrightarrow{*} v_2}{e \vdash (b_1, b_2) \xrightarrow{*} \texttt{pair}(v_1, v_2)} \qquad (52)\quad \frac{e \vdash b_1 \xrightarrow{*} v \qquad e + [x \mapsto v],\ t \vdash b_2 \xrightarrow{*} r}{e \vdash \texttt{let}\ \ x = b_1\ \ \texttt{in}\ \ b_2 \xrightarrow{*} r}$$

$$(53)\quad \frac{e \vdash b \xrightarrow{*} v}{e \vdash \texttt{dynamic}\ \ (b : \sigma) \xrightarrow{*} \texttt{dynamic}(v : T(\sigma, e))}$$

$$(54)\quad Q \vdash v < x \xrightarrow{*} ([x \mapsto v],\ \emptyset) \qquad\qquad (55)\quad Q \vdash i < i \xrightarrow{*} ([\,],\ \emptyset)$$

$$(56)\quad \frac{Q \vdash v_1 < p_1 \xrightarrow{*} (e_1,\ \Gamma_1) \qquad Q \vdash v_2 < p_2 \xrightarrow{*} (e_2,\ \Gamma_2)}{Q \vdash (v, v') < (p, p') \xrightarrow{*} (e \oplus e',\ \Gamma_1 \cup \Gamma_2)}$$

$$(57)\quad \frac{Q \vdash v < p \xrightarrow{*} (e,\ \Gamma) \qquad \overline{\sigma} = \forall \alpha_1 \ldots \alpha_n.\, \overline{\tau}' \qquad \{\alpha_1 \ldots \alpha_n\} \cap BV(Q) = \emptyset}{Q \vdash \texttt{dynamic}(v : \overline{\sigma}) < \texttt{dynamic}(p : \overline{\tau}) \xrightarrow{*} (e,\ \Gamma \cup \{\overline{\tau}' = \overline{\tau}\})}$$

Figure 5: Evaluation rules with mixed quantification in type patterns (error rules similar to rules 19–25 and 30–35 have been omitted)

For dynamic creation, the evaluation of $\texttt{dynamic}(b, \tau)$ now has to transform the static type τ inferred for a before pairing it with the value of a (rule 53). 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. Since existential type variables may depend upon universal variables, an existential variable is actually bound to a type context (a type expression with holes) instead of a simple type expression. We write type contexts as $\lambda\alpha_1 \ldots \alpha_n.\, \overline{\tau}$, where the type variables $\alpha_1 \ldots \alpha_n$ are names for the holes in $\overline{\tau}$. The T function defined below is the evaluation function on types. It maps a type expression τ to a type expression $\overline{\tau}$ not containing Skolem constants, interpreting the Skolem constants according to an environment e.

$$\begin{aligned}
T(\texttt{int}, e) &= \texttt{int} \\
T(\texttt{dyn}, e) &= \texttt{dyn} \\
T(\alpha, e) &= \alpha \\
T(\tau_1 \times \tau_2, e) &= T(\tau_1, e) \times T(\tau_2, e) \\
T(\tau_1 \to \tau_2, e) &= T(\tau_1, e) \to T(\tau_2, e) \\
T(S_\alpha(\tau_1 \ldots \tau_n), e) &= \overline{\tau}[\alpha_1 \leftarrow T(\tau_1, e), \ldots, \alpha_n \leftarrow T(\tau_n, e)] \\
&\qquad \text{if } e(\alpha) = \lambda\alpha_1 \ldots \alpha_n.\, \overline{\tau}
\end{aligned}$$

The T function straightforwardly extends to type schemes:

$$T((\forall\alpha_1 \ldots \alpha_n.\, \tau), e) = \forall\alpha_1 \ldots \alpha_n.\, T(\tau, e) \quad \text{if } \{\alpha_1 \ldots \alpha_n\} \cap Dom(e) = \emptyset$$

For dynamic matching during function application (rule 50), it is no longer possible 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 $\overline{\tau}_1 = \overline{\tau}_2$, where $\overline{\tau}_1$ is the internal type of a dynamic, and $\overline{\tau}_2$ a type pattern. Hence the response m returned by the pattern-matching predicate is either `wrong`, or a pair (e, Γ) of a set e of bindings and a set Γ of equations between types. In addition, the pattern-matching predicate is now parameterized by Q, the quantifier prefix for the matching, becoming $Q \vdash v < p \xrightarrow{*} m$ (rules 54–57). The Q prefix is used in rule 57 to select a trivial instance of the internal type of the dynamic whose free variables do not clash with the variables bound by Q. In a second phase, the function *Solve* is called to resolve the set Γ of equations on types, taking then prefix Q into account. (The next section precisely defines *Solve*.) When the type matching succeeds, *Solve* returns the correct bindings for existentially quantified type variables. Then, the 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 [18], 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 [17]. In our case, the problem consists in checking the validity of propositions of the format

$$q_1\alpha_1 \ldots q_n\alpha_n.(\tau = \tau')$$

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

$$\forall\beta_1 \dots \forall\beta_n \exists\alpha_1 \dots \exists\alpha_n.(\tau = \tau'),$$

and the unification problem to

$$\exists\beta_1 \dots \exists\beta_n \exists\alpha_1 \dots \exists\alpha_n.(\tau = \tau').$$

For the purpose of dynamic matching, we not only want to know whether the proposition $Q.(\tau = \tau')$ holds (where 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 these 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 if 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 τ and τ' if $\theta(\tau) = \theta(\tau')$ and θ is a Q-substitution. If such a substitution exists, τ and τ' are said to be Q-unifiable.*

Clearly, the formula $Q.(\tau = \tau')$ is valid if and only if τ and τ' are Q-unifiable. The following proposition gives a simple way to check whether two terms are Q-unifiable.

Proposition 6 *Two terms τ and τ' are Q-unifiable if and only if τ 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 τ and τ'. Since $\theta(\tau) = \theta(\tau')$, the terms τ and τ' are unifiable. Let μ be their most general unifier. There exists a substitution φ such that $\theta = \varphi \circ \mu$. Hence, 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. It follows that μ is a Q-substitution. □

Proposition 6 shows that if two terms are Q-unifiable, then they possess a most general Q-unifier. Moreover, we immediately get an algorithm to compute the most general Q-unifier of τ and τ': compute the most general unifier of τ and τ', using e.g. Robinson's algorithm, and check that it is a Q-substitution.

We can now define the function *Solve* used in evaluation rule 50. It takes a prefix Q and a set Γ of equations $\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n$. The types $\tau_1 \dots \tau_n$ are trivial instances of the internal types of dynamic values. The types $\tau'_1 \dots \tau'_n$ are the type parts of dynamic patterns. Since the prefix Q binds the variables in $\tau'_1 \dots \tau'_n$ only, we first complete Q to bind the variables in $\tau_1 \dots \tau_n$ also. Let $\alpha_1 \dots \alpha_m$ be the variables free in $\tau_1 \dots \tau_n$. We take $Q' = Q.\exists\alpha_1 \dots \exists\alpha_m$. This existential quantification means that the variables $\alpha_1 \dots \alpha_m$ can be freely instantiated during type matching, since they are universally quantified in the internal types of dynamic values. Because of the renaming constraint in rule 46, the variables $\alpha_1 \dots \alpha_m$ are not already bound by Q.

Let μ be the most general Q'-unifier of $\tau_1 \times \ldots \times \tau_n$ and $\tau'_1 \times \ldots \times \tau'_n$. The 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 defined as follows:

$$\begin{aligned} s(\mu, \alpha_1 \ldots \alpha_n, \epsilon) &= [\,] \\ s(\mu, \alpha_1 \ldots \alpha_n, \forall\alpha.Q) &= s(\mu, \alpha_1 \ldots \alpha_n\alpha, Q) \\ s(\mu, \alpha_1 \ldots \alpha_n, \exists\alpha.Q) &= [\alpha \mapsto (\lambda\alpha_1 \ldots \alpha_n.\mu(\alpha))] + s(\mu, \alpha_1 \ldots \alpha_n, Q) \end{aligned}$$

The transformation s is the run-time counterpart of the Skolemization function S used for static typing in section 3.3. It turns the substitution μ into an evaluation environment that reflects the instantiations performed on the existentially quantified variables during the type matching process.

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 with 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 are therefore given rank ∞. When identifying two variables α and β, the resulting variable is given rank $min(rank(\alpha), rank(\beta))$. Then, binding an existential variable α to a constructed type τ is legal if and only if:

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 2.9), 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 can be recorded on the stack or in registers, as if they were regular variables; without specialization, the unification primitive returns a data structure containing these bindings, and less efficient code is generated to access these bindings.

Specializing unification on one of its arguments is not much harder than specializing matching on its second argument (section 2.9). The techniques developed for the Warren Abstract Machine directly apply [28, 14, 4], 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.

Following the ideas above, the first author has integrated a prototype unification compiler in the CAML system. 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 allows performing unification on some parts of a data structure, and regular pattern-matching on the other parts. Then, dynamic patterns $\texttt{dynamic}(p : \tau)$ are 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 quite efficient. However, we agreed that this efficiency was not worth the extra complication of the compiler, and this prototype was not merged with the CAML release.

4 Assessment

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

4.1 Structured input-output

The CAML system provides the two library functions $\texttt{extern} : \texttt{extern\_channel} \times \texttt{dyn} \rightarrow \texttt{unit}$ and $\texttt{intern} : \texttt{intern\_channel} \rightarrow \texttt{dyn}$, to efficiently write and read data structures on persistent storage, preserving sharing inside the structure.[4] A typical use is, for a separate compiler, to communicate relocation information with its linker, and to save and reload symbol tables representing compiled module interfaces.

The ML type system uses name equivalence for concrete data types. This causes some difficulties with dynamics written to persistent storage: the program reading a dynamic can define some data types with different names than the program that wrote the dynamic, or define data types with the same name, but different structures. There are two solutions to this problem. The first one is to revert to structural equivalence for dynamic type matching. That is, the data type definitions are expanded both in dynamic values and in dynamic type patterns, and structural equivalence is used during matching. The CAML implementation of `intern`/`extern` takes another approach: when writing a dynamic to persistent storage, `extern` also writes the definitions of all concrete data types whose names appear in the type part of the dynamic. When the object is read back, `intern` checks that the reader program defines data types with the same names and structurally similar definitions, and raises an exception if this is not the case. The latter implementation allows for faster dynamic coercion than the former, but it requires information on data type definitions to be available at run-time.

[4] The current implementation of `extern` does not handle functional values, because the CAML compiler generates position-dependent machine code. Even with position-independent code, the persistent objects produced would not be portable across architectures. However, there is no problem with defining `extern` over functions in a byte-coded implementation [5].

In conjunction with character-based communication channels such as Unix sockets, the `intern` and `extern` primitives provide a simple implementation of remote procedure calls (RPC). Here is a sample RPC server for the monomorphic function $\texttt{f} : \tau' \to \tau''$.

```
while true do
  let (inchan, outchan) = accept_connection(...) in
    match intern inchan with
      dynamic(arg : τ') → extern outchan (dynamic (f arg))
done
```

The corresponding client stub function is:

```
let f arg =
  let (inchan, outchan) = establish_connection(...) in
    extern outchan (dynamic arg);
    match intern inchan with
      dynamic(res : τ'') → res
```

The first proposed system supports only remote calls to a monomorphic function. Existential variables, as in the second system, are required to provide a remote interface to a polymorphic function. For instance, here is an RPC server for a sorting function:

```
while true do
  let (inchan, outchan) = accept_connection(...) in
    match intern inchan with
      ∃α. dynamic(order, arg : (α × α → bool) × α list) →
        extern outchan (dynamic(sort order arg))
done
```

4.2 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:

- `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 Coq system [11], a proof development environment based on the Calculus of Constructions, provides the ability to interactively define proof tactics written in CAML, and to apply them on the fly. The CAML macro facility [29, 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 data type representing abstract syntax trees. This constructor embeds constants of arbitrary types inside syntax trees. These constants are produced by compile-time evaluations (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 make 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.

4.3 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 [27] and run-time overloading [23].

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 data type is defined, these functions should 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 functions with type `dyn` → `unit`, to be applied until one succeeds whenever none of the standard cases apply.

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

For instance, assuming that the type α `foo` = `A of` α | `B of` $\alpha \times \alpha$ `foo` has been defined, we could add a printer for type `foo` as follows:

```
new_printer (function
    ∃α. dynamic(A : α foo)  →  print_string "A"
  | ∃α. dynamic(B(x, y) : α 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 ad-hoc generic functions with dynamics has 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 functions such as `equal`: all functions on sets, association lists, ..., would have to operate on dynamics, thus dramatically reducing accuracy of static typing and efficiency of compiled code. Moreover, we cannot statically check that `print` is never 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 these respects. 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 Related work

A number of recent studies have considered languages with dynamic types. Some of these studies deal with the automatic insertion of dynamic creations and coercions that turn "ambivalent" programs (programs that cannot be recognized type-safe nor type-erroneous at compile time) into equivalent programs with run-time type checks [24, 12]. One motivation is to integrate more transparently dynamically-typed and statically-typed objects; another motivation is the efficient compilation of programs written in dynamically-typed languages such as Scheme. In our proposal, we insist on keeping the coercions to and from dynamics explicit in the source code and under the programmer's responsibility: serious programming errors can go unnoticed at compile-time if run-time type checks are automatically inserted, therefore reducing the robustness of programs.

The work on dynamics most closely related to ours is that of Abadi, Cardelli, Pierce, Plotkin and Rémy. Their first paper [2] studies a simply-typed calculus enriched by objects with dynamic type. Dynamics can be coerced to partially unknown types: type patterns in the dynamic coercion construct can contain "pattern variables", that correspond in our second system to existentially-quantified variables. Since there is no polymorphism, type matching is straightforward.

Abadi et al. have recently extended this system to polymorphic types [3]. The main difference between their latest system and our second system is in the format of type patterns in dynamic coercions. Their approach to polymorphic type matching is to allow higher-order pattern variables into type patterns: variables that range over type contexts. For instance, the type pattern $\forall\alpha.\exists\beta.\,\alpha \to \beta$ in our second system corresponds, in their system, to the pattern $\forall\alpha.\,\alpha \to F[\alpha]$, where F is a pattern variable ranging over operators from types to types. Their algebra of type patterns is strictly more expressive than ours: mixed quantification introduces a linear ordering on

the dependencies between existential and universal variables, while pattern variables can express arbitrary dependencies; also, their patterns can select polymorphic functions (e.g. functions that can be applied to integer lists and boolean lists), and these patterns have no equivalent in our second system. On the other hand, mixed quantification has a simple interpretation in first-order logic, and therefore possesses a simple and efficient type matching algorithm; while some ad-hoc restrictions must be put on pattern variables to keep type matching manageable.

6 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, which 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 it is considerably harder to implement. We have presented one promising application of this extension: the use of dynamics for data persistence or interprocess communication. The first proposed system does not allow to operate on persistent data in a generic way, or to perform remote calls to polymorphic functions; the second system supports these operations. More practical experience is needed to determine how expressive a polymorphic type system with dynamics needs to be in order to correctly support these applications.

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. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[3] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 workshop on ML and its applications*, 1992.

[4] Hassan Aït-Kaci. The WAM: a real tutorial. Research report 5, DEC Paris Research Lab, 1990.

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

[6] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[7] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *16th symposium Principles of Programming Languages*, pages 202–212. ACM Press, 1989.

[8] Guy Cousineau and Gérard Huet. The CAML primer. Technical report 122, INRIA, 1990.

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

[10] Roberto Di Cosmo. Type isomorphisms in a type-assignment framework. In *19th symposium Principles of Programming Languages*, pages 200–210. ACM Press, 1992.

[11] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide: version 5.6. Technical report 134, INRIA, 1991.

[12] Fritz Henglein. Dynamic typing. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[13] Xavier Leroy. Polymorphism by name for references and continuations. In *20th symposium Principles of Programming Languages*, pages 220–231. ACM Press, 1993.

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

[15] Harry G. Mairson. Outline of a proof theory of parametricity. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 313–328, 1991.

[16] Michel Mauny. Functional programming in CAML. Technical report 129, INRIA, 1991.

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

[18] Dale Miller. Unification under a mixed prefix. Technical report MS-CIS-91-81, Computer Science Department, University of Pennsylvania, 1991. To appear in *Journal of Symbolic Computation*.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[20] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 367–458. The MIT Press/Elsevier, 1990.

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

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

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

[24] Satish R. Thatte. Quasi-static typing. In *17th symposium Principles of Programming Languages*, pages 367–381. ACM Press, 1990.

[25] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), 1990.

[26] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture 1989*. ACM Press, 1989.

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

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

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