Dynamic Typing in Polymorphic Languages

Martín Abadi, Luca Cardelli
Systems Research Center
Digital Equipment Corporation

Benjamin Pierce
LFCS
University of Edinburgh

Didier Rémy
INRIA Rocquencourt

1 Introduction

Dynamic types are sometimes used to palliate deficiencies in languages with static type systems. They can be used instead of polymorphic types, for example, to build heterogeneous lists; they are also exploited to simulate object-oriented techniques safely in languages that lack them, as when emulating methods with procedures. But dynamic types are of independent value, even when polymorphic types and objects are available. They provide a solution to a kind of computational incompleteness inherent to statically-typed languages, offering, for example, storage of persistent data, inter-process communication, type-dependent functions such as `print`, and the `eval` function.

Hence, there are situations in programming where one would like to use dynamic types even in the presence of arbitrarily advanced type features. In this paper we investigate the interplay of dynamic typing with polymorphism. Our study extends earlier work (see [1]) in allowing polymorphism, but keeps the same basic language constructs (`dynamic` and `typecase`) and the same style.

The interaction of polymorphism and dynamic types gives rise to problems in binding type variables. We find that these problems can be more clearly addressed in languages with explicit polymorphism. Even then, we encounter some perplexing difficulties (as indicated in [1]). In particular, there is no unique way to match the type tagging a dynamic value with a `typecase` pattern. Our solution consists in constraining the syntax of `typecase` patterns, providing static guarantees of unique solutions. The examples we have examined so far suggest that our restriction is not an impediment in practice. This solution applies also to languages with abstract data types, and it extends to languages with subtyping.

Drawing from the experience with explicit polymorphism, we consider languages with implicit polymorphism in the ML style. The same ideas can be used, with some interesting twists. In particular, we are led to introduce tuple variables, which stand for tuples of type variables.

In addition to [1], several recent studies have considered languages with dynamic types [14, 17, 23]. The work most relevant to ours is that of Leroy and Mauny, who define and investigate two extensions of ML with dynamic types. We compare their designs to ours in section 6.

Section 2 is a brief review of dynamic typing in simply-typed languages, based on [1]. Section 3 considers the general case of adding dynamic typing to a language with higher-order polymorphism [11]. An algorithmic formulation of the general framework is obtained in section 3.5 by restricting polymorphism to the second order and placing conditions on the patters used in `typecase` expressions. Sections 4 and 5 discuss abstract data types and subtyping, respectively. Section 6 deals with a language with implicit polymorphism.

2 Review

The integration of static and dynamic typing is fairly straightforward for monomorphic languages. The simplest approach introduces a new base type `Dynamic` along with a `dynamic` expression for constructing values of type `Dynamic` and a `typecase` expression for inspecting them. The typechecking rules for these expressions are:

$$\frac{\Gamma \vdash a \in T}{\Gamma \vdash \texttt{dynamic}(a{:}T) \in \texttt{Dynamic}} \quad \text{(Dyn-I)}$$

$$\frac{\Gamma \vdash d \in \texttt{Dynamic} \qquad \Gamma, x{:}P \vdash b \in T \qquad \Gamma \vdash e \in T}{\Gamma \vdash \texttt{typecase}\ d\ \texttt{of}\ \ (x{:}P)\ b\ \texttt{else}\ e \in T} \quad \text{(Dyn-E)}$$

The phrases $(x{:}P)$ and `else` are *branch guards*; P is a *pattern*—here, just a monomorphic type; b and e are *branch bodies*. For notational simplicity, we have

considered only typecase expressions with exactly one guarded branch and an else clause; typecase involving several patterns can be seen as syntactic sugar for several nested instances of the single-pattern typecase.

In the intended implementation, the compilation of a dynamic is a pair consisting of a value and its type:

```
compile[dynamic(a:A)] =
   "<‘compile[a]’, ‘grab[A]’>"
```

The double quotes here indicate that the result of compile is a run-time structure; single quotes mark substructures to be built at compile time. The keyword "grab" indicates a metalevel shift: a compile-time data structure or routine is inserted into the run-time value. Because of this, Dynamic is particularly easy to implement in a bootstrapped compiler, where the run-time and compile-time structures coincide.

The typecase construct uses the compiler's typematch routine to compare the tag of a given dynamic to the branch guard:

```
compile[typecase d of (x:A) b else e] =
  "let d = ‘compile[d]’ in
   if ‘grab[typematch]’ (‘grab[A]’) (snd(d))
      then push[x=fst(d)]; ‘compile[b]’
      else ‘compile[e]’"
```

In languages with subtyping, it is common to use a subtype test in the typecase construct to give a less restrictive matching rule, allowing a tag to be a subtype of the guard type instead of requiring that the two match exactly.

Constructs analogous to dynamic and typecase have been provided in a number of languages, including Simula-67 [2], CLU [18], Cedar/Mesa [16], Amber [3], Modula-2+ [22], Oberon [25], and Modula-3 [9].

These constructs have surprising expressive power; for example, fixpoint operators can be defined at every type already in a simply typed lambda-calculus extended with Dynamic [1]. Important applications of dynamics include persistence and inter-address-space communication. For example, the following primitives provide input and output of a dynamic value from and to a stream:

```
extern ∈ Writer×Dynamic→Unit
intern ∈ Reader→Dynamic
```

Moreover, dynamics can be used to give a type for an eval primitive [12, 20]:

```
eval ∈ Exp→Dynamic
```

We obtain a much more expressive system by allowing typecase guards to contain pattern variables. For example, the following function takes two Dynamic arguments and attempts to apply the contents of the first (after checking that it is of functional type) to the contents of the second:

```
dynApply =
  λ(df:Dynamic) λ(da:Dynamic)
    typecase df of
      {U,V} (f:U→V)
        typecase da of
          {} (a:U)
            dynamic(f(a):V)
        else ...
    else ...
```

Here U and V are pattern variables introduced by the first guard. In this example, if the arguments are:

```
df = dynamic((λ(x:Int)x+2):Int→Int)
da = dynamic(5:Int)
```

then the typecase guards match as follows:

Tag:	Int→Int
Pattern:	$U \rightarrow V$
Result:	$\{U = \texttt{Int}, V = \texttt{Int}\}$

Tag:	Int
Pattern:	Int
Result:	$\{\}$

and the result of dynApply is dynamic(7 : Int).

A similar example is the dynamic-composition function, which accepts two Dynamic arguments and attempts to construct a Dynamic containing their functional composition:

```
dynCompose =
  λ(df:Dynamic) λ(dg:Dynamic)
    typecase df of
      {U,V} (f:U→V)
        typecase dg of
          {T} (g:T→U)
              dynamic(f∘g:T→V)
        else ...
    else ...
```

3 Explicit Polymorphism

This formulation of dynamic types may be carried over almost unchanged to languages based on explicit polymorphism [11, 21]. For example, the following function checks that its argument df contains a polymorphic function f taking elements of any type into Int. It then creates a new polymorphic function that accepts a type and an argument of that type, instantiates f appropriately, applies it, and squares the result:

```
squarePolyFun =
  λ(df:Dynamic)
    typecase df of
      {} (f:∀(Z)Z→Int)
         λ(W) λ(x:W) f[W](x)*f[W](x)
    else λ(W) λ(x:W) 0
```

Here the type abstraction operator is written λ. Type application is written with square brackets. The types of polymorphic functions begin with $\forall$. For example, $\forall$`(T)T`$\rightarrow$`T` is the type of the polymorphic identity function, λ`(T)` λ`(x:T) x`.

3.1 Higher-order pattern variables

First-order pattern variables, by themselves, do not give us sufficient expressive power in matching against polymorphic types. For example, we might like to generalize the dynamic application example from section 2 so that it can accept a polymorphic function and instantiate it appropriately before applying it:

```
dynApply2ltd =
  λ(df:Dynamic) λ(da:Dynamic)
     typecase df of
        {} (f:∀(Z)?→?)
           typecase da of
              {W} (a:W))
                  dynamic(f[W](a):?)
           else ...
     else dynApply(df)(da)
```

But there is no single expression we can fill in for the domain of `f` that will make `dynApply2ltd` apply to both:

```
df = dynamic
        (λ(Z) λ(x:Z×Z) <snd(x),fst(x)>: ...)
da = dynamic(<3,4>: ...)
```

and:

```
df = dynamic((λ(Z) λ(x:Z→Z) x): ...)
da = dynamic((λ(x:Int) x): ...)
```

Thus we are led to introducing higher-order pattern variables, which range over "pattern contexts"—patterns abstracted with respect to some collection of type variables. These suffice to express polymorphic dynamic application:

```
dynApply2 =
  λ(df:Dynamic) λ(da:Dynamic)
     typecase df of
        {F,G} (f:∀(Z)F(Z)→G(Z))
           typecase da of
              {W} (a:F(W))
                  dynamic(f[W](a):G(W))
           else ...
     else dynApply(df)(da)
```

For example, if:

```
df = dynamic(id:∀(A)A→A)
da = dynamic(3:Int)
```

then the `typecase` expressions match as follows:

Tag:	$\forall$`(A)A`$\rightarrow$`A`
Pattern:	$\forall$`(Z)F(Z)`$\rightarrow$`G(Z)`
Result:	$\{F = \Lambda(X)X, G = \Lambda(X)X\}$
Tag:	`Int`
Pattern:	$F(W)$ (which reduces to W)
Result:	$\{W =$ `Int`$\}$

and the result of the application `dynApply2`$(df)(da)$ is `dynamic`$(id$`[Int]`$(3) :$ `Int`$)$.

3.2 Syntax

We now formalize dynamic types within the context of a higher-order polymorphic λ-calculus, F_ω [11]. The syntax of F_ω with type `Dynamic` is given in Figure 1.

In examples we also use base types, cartesian products, and labeled records in types and patterns, but we omit these in the formal treatment.

We regard as identical any pair of formulas that differ only in the names of bound variables. For brevity, we sometimes omit kinding declarations of the form ": `Type`" and empty pattern-variable bindings. Also, it is technically convenient to write the pattern variables bound by a `typecase` expression as a syntactic part of the pattern, rather than putting them in front of the guard as we have done in the examples. Thus, `typecase e1 of {V}(x:T)e2 else e3` should be read formally as `typecase e1 of (x:{V:Type}T)e2 else e3`.

3.3 Tag Closure

One critical design decision for a programming language with type `Dynamic` is the question of whether type tags must be closed (except for occurrences of pattern variables), or whether they may mention universally bound type variables from the surrounding context.

In the simplest scenario, `dynamic(a:A)` is legal only when A is a closed (but possibly polymorphic) type. Similarly, we would require that the guard in a `typecase` expression be a closed type.

If the closure restriction is not instituted, then types must actually be passed as arguments to polymorphic functions at run time, so that code can be compiled for expressions like:

λ`(X)` λ`(x:X) dynamic(<x,x>:X×X).`

where the type $X \times X$ must be generated at run time. For languages such as ML, where type information is not retained at run time, the closure restriction becomes essential (see section 6). For now, we consider the general case where tags may contain free type variables.

3.4 Definiteness

The most simple-minded formulation of higher-order pattern variables may seem to provide adequate expressive power, but it is not sufficiently constrained to lead to a viable language design. The problem is that there is no guarantee of unique matches of patterns against tags. For example, when the pattern F(`Int`) is matched

K	::=	`Type`	kind of types
	|	$K \rightarrow K$	kinds of type operators
T	::=	A	type variables
	|	$T \rightarrow T$	function types
	|	$\forall(A : K)T$	quantified types
	|	$\Lambda(A : K)T$	type operators
	|	$T(T)$	application of a type operator
	|	`Dynamic`	the dynamic type
P	::=	$\{V_1 : K_1, \ldots, V_n : K_n\}\, T$	patterns
e	::=	x	variables
	|	$\lambda(x : T)e$	abstraction
	|	$e(e)$	application
	|	$\lambda(A : K)e$	type abstraction
	|	$e[T]$	type application
	|	`dynamic`$(e : T)$	tagging
	|	`typecase` e `of` $(x : P)\, e$ `else` e	tag matching

Figure 1: Syntax of F_ω terms

against the tag `Bool`, the pattern variable `F` is forced to be `Λ(X)Bool`. But when the same pattern is matched against the tag `Int`, we find that `F` can be either `Λ(X)X` or `Λ(X)Int`. There is no reasonable way to choose. Worse yet, consider $F(W)$ or $F(W \rightarrow$ `Int`$)$ for a pattern variable `W`.

These problems compel us to introduce restrictions on the form of patterns. We may require that a pattern matches a tag in at most one way at run time, and fail otherwise. But this leads to unpredictable matching failures. Therefore we prefer a stronger condition.

Informally, we want to allow only patterns that match each tag in at most one way. This condition is called *definiteness*. For example, assume that the type variables `A` and `B` and the operator variable `H` are bound in the current context, and consider the following patterns:

Valid: `{V}` `H(V)`

since, at run time, if `V` appears in the expression to which `H` is bound, then matching is the usual for first-order pattern variables; and otherwise matching leaves `V` completely undetermined, and we set it to a new type constant. Patterns of this form are used in many of our examples, so we want to consider them definite even though they may sometimes leave `V` unconstrained.

Invalid: `{F}` `F(A)`

because, for example, it can match the tag $A \rightarrow A$ in four ways, instantiating F to any of: `Λ(C)C→C`, `Λ(C)C→A`, `Λ(C)A→C`, `Λ(C)A→A`.

Valid: `{F}` `∀(X)F(X)`

since, when the scope of X is narrower than that of F, we can only match the tag `∀(Y)Y→Y` by instantiating F to `Λ(C)C→C`.

Valid: `{F}` `(∀(X)F(X))→F(A)`

since the first occurrence of F determines its value.

Valid: `{F,V}` `(∀(X)F(X))→F(V)`

since F can be matched first and then considered "already bound" at the defining occurrence of V.

Invalid: `{F,G}` `∀(X)F(G(X))→G(F(X))`

since neither F nor G can be considered "already bound" unless the other is bound first.

Note that definiteness of patterns cannot be checked locally. For example,

Valid: `{F}` `F(Int)→F(Bool)`

is definite, although neither occurrence of F would be definite in isolation.

The definiteness condition can be formalized, and then one can put a definiteness requirement in the typechecking rule for `typecase`, so that only programs with definite patterns are legal.

Unfortunately, the notion of definiteness does not suggest a typechecking algorithm in any straightforward way. A related problem is that we have no algorithm for the run-time operation of matching patterns against tags. Indeed, it is not known whether the general case of higher-order matching is decidable. Even the second- and third-order cases, whose decidability has been established [15, 7, 8], lead to algorithms too inefficient to be of practical use in implementing `typecase`.

3.5 Second-order polymorphism

To obtain a practical language design, we need a restriction of our general treatment for which efficient typechecking and matching algorithms can be given. We begin by considering the fragment of F_ω with only second-order polymorphism. This restriction is mostly a matter of convenience, and it seems possible that the approach described below applies to the full F_ω.

The syntax of System F (the second-order polymorphic lambda-calculus) with `Dynamic` is given by the following restriction of F_ω with `Dynamic`. We show only the lines that differ.

$$
\begin{array}{rcll}
K & ::= & \ldots & \\
 & | & \texttt{Type}\rightarrow K & \text{type operators} \\
 & & & \\
T & ::= & \ldots & \\
 & | & \forall(A)T & \text{quantified types} \\
 & | & \Lambda(A)T & \text{type operators} \\
 & | & \ldots & \\
 & & & \\
P & ::= & \ldots & \\
 & & & \\
e & ::= & \ldots & \\
 & | & \lambda(A)e & \text{type abstraction} \\
 & | & \ldots &
\end{array}
$$

Here, kinds other than `Type` are used only to specify the functionality of pattern variables; abstractions and applications at the level of types are used only to describe patterns. Since every kind has the form

$$\underbrace{\texttt{Type}\rightarrow\cdots\texttt{Type}}_{n},$$

we can simply say that a pattern variable with this kind has *arity* n.

A pattern $\{V_1 : K_1, \ldots, V_n : K_n\}\ T$ is *ordered* if there is some total ordering $<$ of the pattern variables $V_1, \ldots, V_n$ such that each V_i has a *defining occurrence*, that is, a subterm occurrence $U \equiv V_i\, A_1 \cdots A_p$ in T such that:

1. U does not appear in an argument to a pattern variable V_j where $V_j \geq V_i$ (i.e., there is no occurrence $U' \equiv V_j\, Q$ with U a proper subphrase of U' and $V_j \geq V_i$);
2. V_i is fully applied (i.e., the arity of V_i is p);
3. the A_j's are pairwise distinct; and
4. all of the A_j's have narrower scope than V_i (i.e., $A_j \notin \mathrm{FV}(T)$).

Note that this condition can be checked statically.

Ordered matching is a matching algorithm for ordered patterns; given an ordered pattern, this algorithm instantiates variables according to one of the orders that make the pattern ordered. We believe that ordered patterns are always definite, and that ordered matching is correct, that is, it terminates on every input and always yields the same solution independently of the order of variable instantiations. Hence we replace the definiteness condition with the ordered condition in typechecking, and in evaluation we adopt ordered matching. We omit a detailed description of ordered matching; section 6.3 contains a similar algorithm in a somewhat different context.

4 Abstract Types

The interaction between the use of `Dynamic` and abstract data types gives rise to a puzzling design issue: should the type tag of a dynamically typed value containing an element of an abstract type be matched abstractly or concretely? There are good arguments for both choices:

- Abstract matching protects the identity of "hidden" representation types and prevents accidental matches in cases where several abstract types happen to have the same representation.
- Transparent matching allows a more permissive style of programming, where a dynamically typed value of some abstract type is considered to be a value of a different version of "the same" abstract type. This flexibility is critical in many situations. For example, a program may create disk files containing dynamic values, which should remain usable even after the program is recompiled, or two programs on different machines may want to exchange abstract data in the form of dynamically typed values.

By viewing abstract types formally as existential types [19], we can see exactly where the difference between these two solutions lies and suggest a generalization of existential types that supports both. (Existential types can in turn be coded using universal types; with this coding, our design for dynamic types of the previous sections yields the second solution.)

To add existential types to the variant of F_ω defined in the previous section, we extend the syntax of types and terms as in Figure 2.

The typechecking rules for `pack` and `open` are:

$$\frac{S =_\beta \exists(A : K)\ T \qquad \Gamma \vdash e \in [R/A]T}{\Gamma \vdash (\texttt{pack}\ e\ \texttt{as}\ S\ \texttt{hiding}\ R) \in S} \quad (\textsc{Pack})$$

$$\frac{\Gamma \vdash e \in \exists(A : K)\ S \qquad A \notin \mathrm{FV}(T) \qquad \Gamma, A : K, x : S \vdash b \in T}{\Gamma \vdash (\texttt{open}\ e\ \texttt{as}\ [A, x]\ \texttt{in}\ b) \in T} \quad (\textsc{Open})$$

A typical example where an element of an abstract type is packed into a `Dynamic` is:

$$
\begin{array}{lcll}
T & ::= & \ldots & \\
 & | & \exists(A:K)T & \text{existential types} \\
 & & & \\
e & ::= & \ldots & \\
 & | & \texttt{pack}\ e\ \texttt{as}\ T\ \texttt{hiding}\ T & \text{packing (existential introduction)} \\
 & | & \texttt{open}\ e\ \texttt{as}\ [A,x]\ \texttt{in}\ e & \text{unpacking (existential elimination)}
\end{array}
$$

Figure 2: Extended syntax with existential types

```
let stackpack =
  pack
    push = λ(s:IntList)
              λ(i:Int) cons(i)(s),
     pop = λ(s:IntList) cdr(s),
     top = λ(s:IntList) car(s),
     new = nil
  as Some(X)
       push:X->Int->X,
       pop:X->X, top:X->Int, new:X
  hiding IntList
in
  open stackpack as [Stack,stackops] in
  let dstack =
    dynamic
      (stackops.push(stackops.new)(5):Stack)
  in
     typecase dstack of
       (s:Stack) stackops.top(s)
     else 0
```

Note that this sort of example depends critically on the use of open type tags. As above, open tags must be implemented using run-time types. The evaluation of `pack` must construct a value that carries the representation type.

We have a choice in the evaluation rule for the `open` expression:

- We can evaluate the expression `open` e `as` $[R,x]$ `in` b by replacing the representation type variable R by the actual representation type obtained by evaluating e.
- Alternatively, we can replace R by a new type constant.

Without `Dynamic`, the difference between these rules cannot be detected. But with `Dynamic` we get different behaviors. Since both behaviors are desirable, we may choose to introduce an extended `open` form that provides separate names for the abstract and transparent versions of the representation type:

$$\frac{\Gamma \vdash e \in \exists(H:K)\ S \qquad H \notin \mathrm{FV}(T) \qquad \Gamma, H:K, x:S \vdash [H/R]b \in T}{\Gamma \vdash (\texttt{open}\ e\ \texttt{as}\ [R,H,x]\ \texttt{in}\ b) \in T} \quad (\textsc{Open})$$

In the body of `b`, we can build dynamic values with tags R or H; a `typecase` on the former could investigate the representation type while a `typecase` on the latter could not violate the type abstraction.

5 Subtyping

In simple languages with subtyping (e.g., [3, 9]) it is natural to extend `typecase` to perform a subtyping test instead of an exact match. Consider for example:

```
let dx = dynamic(3:Nat)
in
  typecase dx of
      {} (x:Int) ...
  else ...
```

The first `typecase` branch is taken: although the tag of `dx`, `Nat`, is different from `Int`, we have `Nat`$\leq$`Int`.

Unfortunately, this idea runs into difficulties when applied to more complex languages. In general, there does not exist a most general instantiation for pattern variables when a subtype-match is performed. For example, consider the pattern $V \to V$ and the problem of subtype-matching (`Int`$\to$`Nat`)$\leq(V \to V)$. Both `Int`$\to$`Int` and `Nat`$\to$`Nat` are instances of $V \to V$ and supertypes of `Int`$\to$`Nat`, but they are incomparable. Even when the pattern is covariant there may be no most general match. Given a pattern $V \times V$, there may be a type $A \times B$ such that `A` and `B` have no least upper bound, and so there may be no best instantiation for `V`. This can happen, for example, in a system with bounded quantifiers [6, 10], and in systems where the collection of base types does not form an upper semi-lattice. Linear patterns (where each pattern variable occurs at most once) avoid these problems, but we find linearity too restrictive.

Therefore, we take a different approach that works in general and fits well with the language described in section 3.2. We intend to extend System F with subtyping along the lines of [5]. In order to incorporate also the higher-order pattern variables, we resort to power-kinds [4]. The kind structure of section 3.2 is therefore extended as follows:

$$
\begin{array}{lcll}
K & ::= & \texttt{Type} & \\
 & | & K \to K & \\
 & | & \texttt{Power}(K)(T) & (\text{where } T:K)
\end{array}
$$

Informally, `Power(Type)`(T) is the collection of all subtypes of T, and `Power`$(K \to K')(F)$ is the collection of

all operators of kind $K \to K'$ that are pointwise in subtype relation with F. Subtyping ($\leq$) is introduced by interpreting:

$A \leq B : K$ as $A : \mathtt{Power}(K)(B)$, where $A, B : K$

$F \leq G : (K \to K')$ as $F(X) \leq G(X) : K'$, for all $X : K$, where $F, G : (K \to K')$

The axiomatization of $\mathtt{Power}(K)(T)$ [4] is designed to induce the expected subtyping rules. For example, $A : \mathtt{Power}(A)$ says that $A \leq A$.

As in section 3.5, we limit kinds to appear only in patterns, although we may allow bounded quantifiers $\forall(X \leq T)\, T'$ since they can be handled easily. Because of power-kinds, we can now write patterns such as:

```
typecase dx of
  {V,W≤(V×V)} (x:W) ...
  (that is: {V:Type, W:Power(Type)(V×V)} (x:W))
else ...
```

Each branch guard is used in typechecking the corresponding branch body. The shape of branch guards is $\{V_1 : P_1, \ldots, V_n : P_n\}(x : P)$ where each V_i may occur in the P_j with $j > i$ and in P. This shape fits within the normal format of typing environments, and hence introduces no difficulties for static typechecking.

Next we consider the dynamic semantics of `typecase` in presence of subtyping. The idea is to preserve the previous notion that `typecase` performs exact type matches at run time. Subtyping is introduced as a sequence of additional constraints to be checked at run time only after matching. These constraints are easily checked because, by the time they are evaluated, all the pattern variables have been fully instantiated (perhaps to undetermined types, as discussed in section 3). In the example above, suppose that the tag of `dx` is $(\mathtt{Nat} \times \mathtt{Int}) \times \mathtt{Int}$; then we have the instantiations $W = \mathtt{Nat} \times \mathtt{Int}$ and $V = \mathtt{Int}$. When the matching is completed, we successfully check that $W \leq (V \times V)$.

Some examples will illustrate the additional flexibility obtained with subtyping. First we show how to emulate simple monomorphic languages with subtyping but without pattern variables, where `typecase` performs a subtype test. The first example of this section can be reformulated as:

```
typecase dx of
  {V≤Int} (x:V) f[V](x)
else ...
```

where `f:∀(W≤Int)W→Int`. In this example, the tag of `dx` can be any subtype of `Int`. Note that the assumption `V≤Int` is used statically to typecheck `f[V](x)`.

The next example is similar to `dynApply` in section 2, but the type of the argument can be any subtype of the domain of the function:

```
typecase df of
  {V,W} (f:V→W)
     typecase da of
        {V'≤V} (a:V')
           dynamic(f(a):W)
     else ...
else ...
```

With polymorphic tag types, or with polymorphic pattern types with only first-order pattern variables, nothing new happens except that the matching and subtype tests must be the adequate ones for polymorphism.

The next degree of complexity is introduced by higher-order pattern variables. Just as we had $V' \leq V$, a subtype constraint between two first-order pattern variables, we may have `F≤G:(K→K')` for two higher-order pattern variables `F,G:(K→K')`. As mentioned above, the inclusion is intended pointwise: $F \leq G$ iff `F(X)≤G(X):K'` under the assumption `X:K`.

Another form of dynamic application provides an example of higher-order matches with subtyping:

```
typecase df of
  {F,G:Type→Type,V} (f:∀(Z≤V)F(Z)→G(Z))
     typecase da of
       {W≤V} (a:F(W))
          dynamic(f[W](a):G(W))
     else ...
else...
```

Finally, dynamic composition calls for a constraint of the form $G' \leq G$:

```
typecase df of
  {G,H:Type→Type} (f:∀(X)G(X)→H(X))
    typecase dg of
      {F:Type→Type,G'≤G:(Type→Type)}
                          (g:∀(Y)F(Y)→G'(Y))
         dynamic
           ((λ(Z) f[Z]∘g[Z]):∀(Z)F(Z)→H(Z))
    else ...
else ...
```

This example generalizes to functions of bounded polymorphic types, such as `∀(X≤A)G(X)→H(X)`.

6 Implicit Polymorphism

In this section we investigate dynamics in an implicitly typed language, namely ML. First we show that the general treatment of dynamics for explicitly typed languages directly applies to ML, providing explicitly tagged dynamics in an untyped language. This solution is not in the spirit of ML, and all the rest of the section will be devoted to the study of implicitly tagged dynamics in ML.

In the obvious extension of ML, types can still be inferred for all constructs but dynamics; the user needs

to provide type information when creating or reading dynamics. For instance, let us consider the program:

```
twice = dynamic
  (λ(f) λ(x) f(f x):∀(A)(A→A)→(A→A))
```

First, the type scheme `∀(A)(A→A)→(A→A)` is inferred for `λ(f) λ(x) f(f x)` as if it were to be let-bound. Then we check that this type scheme has no free variables and is more general than the required tag `∀(A)(A→A)→(A→A)`. Conversely, when the extraction of a value from a dynamic succeeds, it is given the type scheme of its tag as if it had been let-bound. Thus, all instances of the value can be used with different instances of the tag as in

```
foo = λ(df)
  typecase df of
     (f:∀(A)(A→A)→(A→A)) <f succ, f not>
  else ...
```

where *succ* and *not* are the successor function on integers and the negation function on booleans.

This works perfectly. However, it requires explicit type information in dynamic patterns, which is not in the spirit of ML. Since the ML typechecker can infer most general types for expressions, one would expect the compiler to tag dynamic values with their principal types. For instance, the user writes

```
twice = dynamic(λ(f) λ(x) f (f x))
```

and the dynamic is tagged with `∀(A)(A→A)→(A→A)`.

However, there is a difficulty with the program

```
apply = dynamic(λ(f) λ(x) f x)
```

Should the dynamic's tag be `∀(A,B)(A→B)→(A→B)` or `∀(B,A)(A→B)→(A→B)`? As `typecase` is defined, it succeeds if the tag exactly matches the pattern, including quantifiers. With implicit tagging, the order of quantifiers should not matter.

Moreover, since the tag of *twice* is more general than the pattern of the function *foo*, an ML programmer would probably expect that *twice* can be passed to *foo*. This is also justified by the fact that the typechecker could have built a dynamic with a weaker tag, and the typecase would have succeeded. That is, in ML, the `typecase` would be expected to succeed if an instance of the tag matches the pattern. This principle is called *tag instantiation*. Dynamics with tag instantiation but no pattern variables have been implemented in the language CAML [24]. The dynamics studied by Leroy and Mauny [17] have tag instantiation and first-order pattern variables. First-order pattern variables are not powerful enough to type some reasonable examples, such as the `applyTwice` function shown later. Below we describe a version of dynamics for ML with tag instantiation and second-order pattern variables.

6.1 Tuple variables

Tag instantiation and second-order pattern variables do not fit well together. The difficulty comes from the merging of two features:

- As in the pattern `{F} (f:∀(A)F(A)→A)`, second-order pattern variables may depend on universal variables.
- Tag instantiation requires that if a `typecase` succeeds, then it also succeeds for a dynamic with an argument that has a more general tag. The tag `∀(A)(A×A)→A` matches the previous pattern. So should the tag `∀(A,B)(A×B)→A`. But `F` is not supposed to depend on `B`!

Because of tag instantiation, polymorphic pattern variables may always depend on more variables than the ones explicitly mentioned. We capture all variables that appear in the tag but that do not correspond to variables in the pattern into a tuple of variables `P`. The dependence of pattern variable `F` on all universal variables is written `F(P)`, even though the exact set of variables in `P` is not statically known. The tuple variable `P` will be dynamically instantiated to the tuple of all variables of the tag not matched with variables of the pattern. In particular, if the pattern is `F(P;A)`, then `P` will never contain `A`.

For instance, the pattern `{F} (f:∀(A)F(A)→A)` should be written `{F} (f:∀(A)F(P;A)→A)`, so that tag instantiation is possible. The tag `∀(A)(A×A)→A` matches this pattern for an empty tuple. The tag `∀(A,B)(A×B)→A` matches it for a one-element tuple, namely (B).

Tuple variables bound in different patterns may be instantiated to tuples with different numbers of variables, as in the example just given. Because of such size considerations, it is not always possible to use a tuple variable as argument to an operator, since it may expect an argument of different size. We use tuple sorts in order to guarantee that type expressions are well formed. Formally, a `typecase` with explicit information should be written, for instance:

```
{p Tuple,F:p→Type→Type}
    (f:∀(P:p,A:Type)F(P;A)→A)
```

where $F(A_0; A_1, \ldots, A_n)$ stands for a fully applied pattern variable $F(A_0)(A_1)\ldots(A_n)$; this notation reminds that the first argument A_0 is a tuple. The sort variable `p` is to be bound at run time. However, it is not necessary to write the sorts in programs since a typechecker can easily infer them.

6.2 Description of the language

We assume given a denumerable collection of tuple sorts, written π, π', etc., and a sort *Type*. Then the

sorts are:

$$\begin{array}{ll} k ::= \pi \mid \mathit{Type} & \text{atomic sorts} \\ K ::= k \mid k \to K & \text{sorts} \end{array}$$

Types are:

$$\begin{array}{ll} T ::= \texttt{Dynamic} \mid A \mid T(T) \mid T \to T & \text{types} \\ P ::= \{A_1, \ldots, A_n\}T & \text{patterns} \\ S ::= \forall(A_1 : K_1, \ldots, A_n : K_n)\ T & \text{type schemes} \end{array}$$

In traditional ML style, we have left quantifiers implicit in types and hence in patterns. The formation of types, patterns, and contexts is controlled by judgements of the form:

$$\begin{array}{ll} \Gamma \vdash T \in K & \text{type } T \text{ is of sort } K \text{ in } \Gamma \\ \Gamma \vdash S \in K & \text{type schema } S \text{ is of sort } K \text{ in } \Gamma \\ \Gamma \vdash P \in K & \text{pattern } P \text{ is of sort } K \text{ in } \Gamma \end{array}$$

where contexts are:

$$\begin{array}{ll} \Gamma ::= \emptyset & \text{empty context} \\ \quad \mid \Gamma, \pi & \text{tuple sort declaration} \\ \quad \mid \Gamma, F : K & \text{symbol declaration} \end{array}$$

Formation rules ensure that type variables are always bound in the proper context, with a sort consistent with their uses. For instance, we have the rule:

$$\frac{\Gamma \vdash T : k \qquad \Gamma \vdash T' : k \to K}{\Gamma \vdash T'(T) : K} \quad \text{(Sort-App)}$$

In examples only, we help the reader by using different letters for variables of different sorts: type variables of tuple sorts are written `P` and `Q` and type variables of sort Type are written `A`, `B`, etc. We also use `F`, `G`, and `H` for pattern variables.

Patterns are pairs of a set of pattern variables $\{V_1, \ldots, V_n\}$ and a type T. They are well formed if the signature of all V_i's if of the form $\pi \to K$ for the same tuple sort π. The exact rule for pattern formation is:

$$\frac{\begin{array}{c} \pi, A_0, A_1, \ldots, A_n \notin \Gamma \\ \Gamma, \pi, V_1 : \pi \to K_1, \ldots, V_n : \pi \to K_n, \\ A_0 : \pi, A_1 : \mathit{Type}, \ldots, A_n : \mathit{Type} \vdash T \in \mathit{Type} \end{array}}{\begin{array}{l} \Gamma, \pi \vdash \{V_1 : \pi \to K_1, \ldots, V_n : \pi \to K_n\}\ T \\ \qquad \in \forall(A_0 : \pi, A_1 : \mathit{Type}, \ldots, A_n : \mathit{Type})\ T \end{array}}$$

In particular, there is exactly one pattern variable of tuple sort per pattern.

Again, we would like to guarantee definiteness of patterns, and we impose the sufficient condition that patterns be ordered. Ordered patterns are those that satisfy the conditions given in section 3.5, and in addition all non-pattern free variables of sort Type must appear at least once outside of all pattern variables in the pattern. In the context of ML, our definiteness requirement is reminiscent of the type-explication restriction imposed on signatures in Standard ML (see section 7.7 of [13]).

Pattern variables are bound at the beginning of the `typecase`, and their scope is the `typecase` in which they have been introduced. All other free variables are bound at the beginning of the pattern and their scope is the pattern.

Expressions are:

$$\begin{array}{l} e ::= x \mid \lambda(x)\ e \mid e\ e \\ \quad \mid \texttt{dynamic}(e) \\ \quad \mid \texttt{typecase}\ e\ \texttt{of} \\ \qquad \{V_1 : K_1, \ldots, V_n : K_n\}\quad (x : P)\ e\ \texttt{else}\ e \end{array}$$

Type inference Pattern variables behave as local type symbols in ML. Typechecking with local type symbols implies an extension of judgement contexts in order to control the scope of type symbols:

$$\begin{array}{ll} \Gamma ::= \ldots & \\ \quad \mid \Gamma, x : S & \text{variable type assignment} \end{array}$$

The typing judgements are $\Gamma \vdash e : T$.

The "instance" rule of ML becomes:

$$\frac{\Gamma \vdash T : \mathit{Type} \qquad \begin{array}{c} e : S \in \Gamma \\ T \text{ is an instance of } S \end{array}}{\Gamma \vdash e \in T} \quad \text{(Inst)}$$

It says that instances have to be well formed in the current context, which prevents us from using local symbols out of their scope.

Since we do not want to carry types at run time, we require that the types of values to become dynamic be closed, and then tags can be statically compiled.

$$\frac{\Gamma \vdash e \in S \qquad S \text{ is closed}}{\Gamma \vdash \texttt{dynamic}(e) \in \texttt{Dynamic}} \quad \text{(Dyn-I)}$$

This rule may destroy the principal typing property of ML. If the principal type of an expression e is S and S is not closed, then typing $\texttt{dynamic}(e)$ requires that free variables of S are instantiated by ground types. However, the set of closed instances of a principal type that is not closed does not have a principal element.

We want to avoid such situations, since the nonexistence of a principal type corresponds to an ambiguity concerning the tag that a dynamic value should carry. Therefore, we say that a program is not well typed if it has no principal typing derivation.

Type inference is realized by the same algorithm as in ML but delaying tag-closure checking to the end of typechecking (by gathering free variables of types of dynamic values in a list, for instance). If one of these variables is still free at the end of typechecking, then there exists no principal derivation, and the program is not well typed.

The rule for `typecase` is:

$$\frac{\begin{array}{c}\Gamma \vdash d \in \texttt{Dynamic} \qquad \Gamma \vdash e' \in B \\ \Gamma, \pi \vdash \{V_1 : K_1, \ldots, V_n : K_n\} T \in S \\ \Gamma, \pi, V_1 : K_1, \ldots, V_n : K_n, x : S \vdash e \in B\end{array}}{\begin{array}{c}\Gamma, P : \pi \vdash \texttt{typecase}\ d\ \texttt{of} \\ \{V_1, \ldots, V_n\}\ \ (x\texttt{:}T)\ \ e\ \ \texttt{else}\ e' \in B\end{array}}$$

The other rules of ML are unchanged.

6.3 Evaluation

Compilation is easily decomposed into two phases. The first phase translates ML into a variant, called ML$^-$, where dynamics are explicitly typed; this translation requires a bit of inference. ML$^-$ differs form ML only in its `dynamic` construct:

$$e ::= \ldots \mid \texttt{dynamic}(e : S)$$

and its typing rule:

$$\frac{\Gamma \ \vdash\ a \in S \qquad S \text{ is closed}}{\Gamma \ \vdash\ \texttt{dynamic}(a : S) \in \texttt{Dynamic}} \quad \text{(Dyn-I)}$$

The translation of an ML program e into ML$^-$ is any ML$^-$ program M whose principal type derivation is also a principal type derivation for e. This defines M uniquely (types being equal up to alpha-conversion). The type reconstruction algorithm is a trivial adaptation of the usual type inference algorithm. The semantics of an ML program e is the semantics of its translation into ML$^-$.

The evaluation rules are mostly standard. The only interesting one is for `typecase`, as it involves new methods for matching and pattern-variable instantiation.

Matching is not quite as usual, since it allows tag instantiation, and it also has to deal with tuple variables. Its inputs are a pattern $\{V_1 : K_1, \ldots, V_n : K_n\}\ \ T$ and a tag, that is, a closed type $\forall(\alpha_1, \ldots, \alpha_n)\ \ \tau$. The pattern variables are the V_i's, and the universal variables are the remaining free variables of T. The set of variables that occur in the tag (the α_i's) can increase during tag instantiation. The algorithm returns a substitution μ with domain the pattern variables, such that there exists a substitution μ' with domain the tag variables, and with $\mu'(\tau) = \mu(T)$.

We describe the algorithm as transformations on sets of unification constraints called unificands; the transformations keep unchanged the set of substitutions that satisfy the constraints. The substitutions that we consider can instantiate both pattern and tag variables, but not universal variables.

The metavariable T still stands for any type, and τ stands for a type that does not contain pattern variables. The atomic constraints are pairs, $T \doteq \tau$ or $\tau \doteq T$. The pairs $T \doteq \tau$ and $\tau \doteq T$ are considered equal. A substitution μ is a solution of an atomic constraint if it unifies both sides. In addition, the constant $\bot$ is used to represent failure; it is the atomic constraint with no solution.

In general, a unificand U is an atomic constraint, the conjunction of two unificands $U' \wedge U''$, or the existential unificand $\exists\alpha.U'$. The solution set of $U' \wedge U''$ is the intersection of the solution sets of U' and U''. The solution set of the existential unificand $\exists\alpha.U'$ is the set of solutions of U' restricted to variables distinct from α. We identify unificands up to: commutativity and associativity of conjunction, renaming of variables bound by $\exists$'s, exchange of consecutive $\exists$'s, and removal of vacuous $\exists$'s.

Two unificands U and V are equivalent if they have the same set of solutions. This obviously defines an equivalence relation on unificands, and in fact a congruence.

We reduce the original matching problem to that of finding the solutions of the unificand $\exists \mathrm{FV}(\tau).(T \doteq \tau)$. In order to solve this problem, we now give a list of equivalences between unificands—the unificand on top is always equivalent to the one at the bottom. Tag variables are written α; C and C' are constant symbols, and always occur fully applied; and X is either a universal variable A or a pattern variable V. The set of all variables is $\mathcal{V}$.

- Decomposition

$$\frac{C(T_i) \doteq C(\tau_i)}{\bigwedge(T_i \doteq \tau_i')} \qquad \frac{C(T_i) \doteq C'(\tau_j)}{\bot} \quad C \neq C'$$

- Instantiation

$$\frac{C(T_i) \doteq \alpha}{\exists\alpha_i.\Big(\alpha \doteq C(\alpha_i) \wedge \bigwedge(T_i \doteq \alpha_i)\Big)}$$

$$\frac{V(A_0; \alpha_1, \ldots, \alpha_n) \doteq \tau}{V \doteq \Lambda(A_0; \alpha_1, \ldots, \alpha_n).\tau}$$

- Propagation

$$\frac{X \doteq \tau \wedge M}{X \doteq \tau \wedge [\tau/X]M}$$

$$\frac{\alpha \doteq \tau}{\bot} \qquad \alpha \notin \mathrm{FV}(\tau), \tau \notin \mathcal{V}$$

- Universal-variable restriction

$$\frac{A_i \doteq \alpha \wedge A_j \doteq \alpha}{\bot} \qquad \frac{A \doteq \tau}{\bot} \qquad \tau \notin \mathcal{V}$$

- Existential simplifications

$$\frac{U \wedge (\exists\alpha.U')}{\exists\alpha.(U \wedge U')} \qquad \alpha \notin \mathrm{FV}(U)$$

$$\frac{\exists\alpha.(\alpha \doteq \tau \wedge U)}{\exists\alpha.U} \qquad \alpha \notin \mathrm{FV}(\tau) \cup \mathrm{FV}(U)$$

- Trivial constraints

$$\frac{\alpha \doteq \alpha \wedge U}{U} \qquad \frac{\bot \wedge U}{\bot}$$

These equivalences can be used as rewriting rules. All rules are oriented from top to bottom; one step of rewriting is the application of exactly one rule; applying the rules in any order always terminates. When successful, this process produces canonical unificands of the form:

$$\exists \alpha_k . \Big(\bigwedge (A_i = \alpha_i) \wedge \bigwedge (V_j = \tau_j) \Big)$$

A unificand that cannot be reduced and that is not yet in canonical form is either $\bot$ or contains a constraint $V(A_0; T_1, \ldots, T_n) \doteq \tau$. The ordered condition on patterns prevents the latter (as the second instantiation rule would apply to one of the constraints). Hence, for ordered patterns, rewriting always produces either a canonical unificand or $\bot$.

Because of the form of the rules, the matching is unitary, and all solutions are equal up to renaming of the α_k's. The unique tuple variable that appears in all the τ_j can be bound to the tuple (α_k), and its size bound to the tuple sort.

6.4 Related work

The work on dynamics most closely related to ours is that of Leroy and Mauny [17]. Our system can be seen as an extension of their system with "mixed quantification."

Instead of introducing a *typecase* statement, Leroy and Mauny merge dynamic elimination with the usual case statement of ML. If we ignore this difference, their dynamic patterns have the form $\mathcal{Q}A$ where A is a type and $\mathcal{Q}$ a list of existentially or universally quantified variables.

For instance,

```
∀(A)∃(F)∀(B)∃(G) (v:T(A,F,B,G))
```

is a pattern of their system. The existentially-quantified variables play the role of our pattern variables. The order of quantifiers determines the dependencies among quantified variables. Thus, the pattern above can be rephrased:

```
∃(F)∃(G)∀(A)∀(B) (v:T(A,F(A),B,G(A,B)))
```

Writing quantifiers in our patterns explicitly (for ease of comparison), the equivalent pattern in our system is:

```
{F,G} (v:∀(P,A,B)T(A,F(P;A),B,G(P;A,B)))
```

With the same approach, in fact, we can translate all of their patterns into equivalent patterns in our system, preserving the intended semantics.

On the other hand, there does not seem to be a translation from our language to theirs. They have no pattern equivalent to our pattern:

```
{F,G} (v:∀(P,A,B)T(A,F(P;A),B,G(P;B)))
```

because the quantifiers in the prefix of their patterns are in linear order, and hence it is not possible to have the "parallel" dependencies of `F` on `A` and of `G` on `B`. We can obtain a system intermediate between theirs and ours by leaving tuple variables implicit, and there we would rewrite the pattern above:

```
{F,G} (v:∀(A,B)T(A,F(A),B,G(B)))
```

However, we believe that explicit tuple variables are useful, since they allow examples like the `applyTwice` function:

```
let applyTwice =
  λ(df) λ(dxy)
    typecase df of
      {F,F'} (f:F(P)→F'(P))
        typecase dxy of
          {G,H} (x,y:F(G(Q)) × (F(H(Q))))
              f x, f y
        else ...
    else ...
```

This cannot be expressed in our intermediate system, nor in systems with just type quantifiers, such as Leroy and Mauny's.

References

[1] 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, April 1991.

[2] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[3] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages.* Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[4] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[5] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, number 526 in Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, September 1991.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[7] Gilles Dowek. A second order pattern matching algorithm in the cube of typed λ-calculi. In *Proceedings of Mathematical Foundation of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 151–160. Springer Verlag, 1991. Also Rapport de Recherche INRIA, 1992.

[8] Gilles Dowek. Third order matching is decidable. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992. To appear.

[9] Greg Nelson (ed.). *Systems Programming in Modula-3*. Prentice Hall, 1991.

[10] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[11] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[12] Mike Gordon. Adding Eval to ML. Personal communication, circa 1980.

[13] Robert Harper, Robin Milner, and Mads Tofte. *Commentary of Standard ML*. The MIT Press, 1991.

[14] Fritz Henglein. Dynamic typing. In *ESOP*, 1992.

[15] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[16] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[17] Xavier Leroy and Michel Mauny. Dynamics in ML. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[18] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[19] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

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

[21] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag Lecture Notes in Computer Science 19.

[22] Paul Rovner. On extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[23] Satish R. Thatte. Quasi-static typing (preliminary report). In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[24] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Ascander Suárez. The CAML reference manual. Research report 121, INRIA, Rocquencourt, September 1990.

[25] Niklaus Wirth. From Modula to Oberon and the programming language Oberon. Technical Report 82, Institut für Informatik, ETH, Zurich, 1987.