January 26, 1994

SRC Research Report 120

Dynamic Typing in Polymorphic Languages

Martín Abadi, Luca Cardelli,
Benjamin Pierce, and Didier Rémy

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Dynamic Typing
in Polymorphic Languages

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy

June 4, 1994

Benjamin Pierce is at the Department of Computer Science of the University of Edinburgh. Part of his work was done at INRIA Rocquencourt and part at the School of Computer Science, Carnegie Mellon University. Didier Rémy is at INRIA Rocquencourt.

A preliminary version of this material appeared as [2].

Authors' Abstract

There are situations in programming where some dynamic typing is needed, even in the presence of advanced static type systems. We investigate the interplay of dynamic types with other advanced type constructions, discussing their integration into languages with explicit polymorphism (in the style of system F), implicit polymorphism (in the style of ML), abstract data types, and subtyping.

Contents

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 advanced static type systems. In this paper we investigate the integration of dynamic types into languages with explicit polymorphism (in the style of system F [10]), implicit polymorphism (in the style of ML [16]), abstract data types, and subtyping. Our study extends earlier work [1], but keeps the same general approach and the same basic language constructs: `dynamic`, for tagging a value with its type, and `typecase`, for comparing a type tag with a pattern and branching according to whether they match.

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 tag of a dynamic value with a `typecase` pattern. Our solution consists in constraining the syntax of `typecase` patterns, thus providing static guarantees of unique solutions. The examples we have examined so far suggest that our restriction is not an impediment in practice.

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.

The treatment of abstract data types does not present any new typing or matching difficulties. Instead, it raises an interesting question: whether the type tag of a dynamically typed value should be matched abstractly or concretely (that is, using knowledge of the value's actual run-time type). We explore the consequences of both choices.

Subtyping is exploited in combination with dynamic types in languages with restricted `typecase` patterns, such as Simula-67 [3] and Modula-3 [8]. There, a tag matches a pattern if it is a subtype of the pattern. This sort of matching does not work out well with more general `typecase` patterns. We find it preferable to match type tags against patterns exactly, and then perform explicitly prescribed subtype tests.

In addition to [1], several recent studies consider languages with dynamic types [12, 14, 21]. 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 4.

Section 2 is a brief review of dynamic typing in simply typed languages, based on [1]. Section 3 considers the addition of dynamic typing to a language with explicit polymorphism [10]. Section 4 then deals with a language with implicit polymorphism. Sections 5 and 6 discuss abstract data types and subtyping, respectively. We conclude in section 7.

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` (informally called "dynamics") and a `typecase` expression for inspecting them. The typechecking rules for these expressions are:

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

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

The phrases $(x{:}P)$ and `else` are *branch guards*; P is a *pattern*—here, just a monomorphic type; b and c are *branch bodies*. For notational simplicity, we have considered only `typecase` expressions with exactly one guarded branch and an `else` clause; a `typecase` involving several patterns can be seen as syntactic sugar for several nested instances of the single-pattern `typecase`.

In the most direct implementation, the semantics of a dynamic is a pair consisting of a value and its type. The semantics of `typecase d of (x:A) b else e` is then: break `d` into a value `c` and a type `C`, and if `C` matches `A` then bind `x` to `c` and execute `b`, otherwise execute `e`. In the simplest version of this semantics, `C` matches `A` if they are identical; in languages with subtyping, it is common to allow `C` to be a subtype of `A` instead.

Constructs analogous to `dynamic` and `typecase` have appeared in a number of languages, including Simula-67 [3], CLU [15], Cedar/Mesa [13], Amber [4], Modula-2+ [20], Oberon [23], and Modula-3 [8]. These constructs have surprising expressive power; for example, fixpoint operators can already be defined at every type 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 might 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 [11, 18]:

```
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 dynamics 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:	$\mathtt{Int}\rightarrow\mathtt{Int}$
Pattern:	$\mathtt{U}\rightarrow\mathtt{V}$
Result:	$\{\mathtt{U}=\mathtt{Int}, \mathtt{V}=\mathtt{Int}\}$
Tag:	`Int`
Pattern:	`Int`
Result:	$\{\}$

and the result of `dynApply` is $\mathtt{dynamic}(7:\mathtt{Int})$.

A similar example is the dynamic-composition function, which accepts two dynamics as 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
          {W} (g:W→U)
              dynamic(f ∘ g:W→V)
        else ...
    else ...
```

3 Explicit Polymorphism

This formulation of dynamic types may be carried over almost unchanged to languages based on explicit polymorphism [10, 19]. For example, the following function checks that its argument `df` contains a polymorphic function `f` taking lists to lists. It then creates a new polymorphic function that accepts a type and a list `x` of values of that type, instantiates `f` appropriately, and applies `f` to the reverse of `x`:

```
λ(df:Dynamic)
  typecase df of
    {} (f:∀(Z) List(Z)→List(Z))
       λ(Y) λ(x:List(Y)) f[Y](reverse[Y](x))
  else λ(Y) λ(x:List(Y)) x
```

Here `List` and `reverse` have the obvious meanings; they are not primitives of the language treated below, but can be encoded. The type abstraction operator is written λ. Type application is written with square brackets. The types of polymorphic functions begin with $\forall$. For example, $\forall(\mathtt{X})\mathtt{X}\to\mathtt{X}$ is the type of the polymorphic identity function, $\lambda(\mathtt{X})\ \lambda(\mathtt{x} : \mathtt{X})\ \mathtt{x}$.

3.1 Higher-order pattern variables

First-order pattern variables, by themselves, do not appear to 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:

```
dynApply2try =
  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 `dynApply2try` 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): ...)
```

In the former case, the expected type for `f` is `∀(X)(X×X)→(X×X)`; in the latter case, it is `∀(X)(X→X)→(X→X)`. These two types are incompatible.

Thus we are led to introducing higher-order pattern variables. A higher-order pattern variable ranges over *pattern contexts*—patterns abstracted with respect to some collection of type variables. With higher-order pattern variables, we can 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:∀(X)X→X)
da = dynamic(3:Int)
```

then the `typecase` expressions match as follows:

Tag:	`∀(X)X→X`
Pattern:	`∀(Z)F(Z)→G(Z)`
Result:	$\{\mathtt{F} = \Lambda(\mathtt{X})\ \mathtt{X}, \mathtt{G} = \Lambda(\mathtt{X})\ \mathtt{X}\}$

Tag:	`Int`
Pattern:	`F(W)` (which reduces to `W`)
Result:	$\{\mathtt{W} = \mathtt{Int}\}$

and the result of the application

$$\mathtt{dynApply2(df)(da)}$$

is

$$\mathtt{dynamic(id[Int](3) : Int)}$$

Following standard notational conventions, we write Λ`(X)X` for the identity function on types, reserving lowercase letters for expressions at the level of terms and λ for term-level abstractions.

It is now easy to deal with the two inputs given above. If

```
df = dynamic (λ(Z) λ(x:Z×Z) <snd(x),fst(x)>:
             ∀(X)(X×X)→(X×X)))
da = dynamic(<3,4>:Int×Int)
```

then the match is:

Tag:	$\forall(\mathtt{X})(\mathtt{X}\times\mathtt{X})\to(\mathtt{X}\times\mathtt{X})$
Pattern:	$\forall(\mathtt{Z})\mathtt{F}(\mathtt{Z})\to\mathtt{G}(\mathtt{Z})$
Result:	$\{\mathtt{F} = \Lambda(\mathtt{X})\ \mathtt{X}\times\mathtt{X}, \mathtt{G} = \Lambda(\mathtt{X})\ \mathtt{X}\times\mathtt{X}\}$

Tag:	`Int`
Pattern:	$\mathtt{F}(\mathtt{W})$ (which reduces to $\mathtt{W}\times\mathtt{W}$)
Result:	$\{\mathtt{W} = \mathtt{Int}\times\mathtt{Int}\}$

If

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

then the match is:

Tag:	$\forall(\mathtt{X})(\mathtt{X}\to\mathtt{X})\to(\mathtt{X}\to\mathtt{X})$
Pattern:	$\forall(\mathtt{Z})\mathtt{F}(\mathtt{Z})\to\mathtt{G}(\mathtt{Z})$
Result:	$\{\mathtt{F} = \Lambda(\mathtt{X})\ \mathtt{X}\to\mathtt{X}, \mathtt{G} = \Lambda(\mathtt{X})\ \mathtt{X}\to\mathtt{X}\}$

Tag:	`Int`
Pattern:	$\mathtt{F}(\mathtt{W})$ (which reduces to $\mathtt{W}\to\mathtt{W}$)
Result:	$\{\mathtt{W} = \mathtt{Int}\to\mathtt{Int}\}$

3.2 Syntax

We now present dynamic types in the context of a second-order polymorphic λ-calculus, system F. The syntax of F with type `Dynamic` (including some third-order constructs used in patterns) is given in Figure 1. In examples we also use base types, cartesian products, and lists 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 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`.

$$
\begin{array}{rll}
K ::= & \texttt{Type} & \text{the kind of types} \\
\\
T ::= & Z & \text{type variables} \\
| & F & \text{first-order pattern variable} \\
| & T \to T & \text{function types} \\
| & \forall (Z : K)\ T & \text{quantified types} \\
| & F(T^n) \qquad (n > 0) & \text{application of a type operator} \\
| & \texttt{Dynamic} & \text{the dynamic type} \\
\\
J ::= & K & \text{kind of simple pattern types} \\
| & K^n \to K \quad (n > 0) & \text{functional kinds} \\
\\
P ::= & \{F_1 : J_1, \ldots, F_n : J_n\}\ T & \text{patterns} \\
\\
a ::= & x & \text{variables} \\
| & \lambda(x : T)\ a & \text{abstraction} \\
| & a(a) & \text{application} \\
| & \lambda(Z : K)\ a & \text{type abstraction} \\
| & a[T] & \text{type application} \\
| & \texttt{dynamic}(a : T) & \text{tagging} \\
| & \texttt{typecase}\ a\ \texttt{of}\ (x : P)\ a\ \texttt{else}\ a & \text{tag matching}
\end{array}
$$

Figure 1: Syntax for system F with `Dynamic`

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 type. (The type `A` may be polymorphic, of course, so long as all the type variables it mentions are also bound within `A`; for example, $\forall$(`Z`) `Z` is a legal tag.) 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 where type information is not retained at run time, such as ML, the closure restriction becomes essential (see section 4). For now, we consider the unrestricted case, where tags may contain free type variables.

3.4 Definiteness and matching

When pattern variables may range over functions on types, there is in general no guarantee of unique matches of patterns against tags. For example, when the pattern `F(Int)` is matched 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→Int)` for a pattern variable `W`.

Two sorts of solutions come to mind:

- At run time, we may look for matches and fail if none or more than one exist. Unfortunately, failures could be somewhat unpredictable.
- At compile time, we may allow only patterns that match each tag in at most one way. This condition on patterns is called *definiteness* in a preliminary version of this work [2]. As definiteness seems hard to decide at compile time, an approximation to definiteness may be used instead.

As in [2], we choose a compile-time solution. We propose a condition on patterns sufficient to guarantee definiteness ((1) and (2), below), in combination with an appropriate definition of matching ((3), below). This condition is more restrictive than that of [2], and in some cases it may entail some code duplication. On the other hand, it is easier to describe, it suffices for our examples, and in general it does not seem to affect expressiveness.

Our condition on patterns is:

1. each pattern variable introduced in a pattern is used in the same pattern;
2. in each of these uses, the arguments of the pattern variable are distinct type variables bound in the same pattern (not pattern variables).

Note that as far as inner `typecase` expressions are concerned, pattern variables of outer `typecase` expressions are just constants, and they may appear in any positions where constants may appear.

For example, we allow:

$\{$`F` : `Type→Type`$\}\forall$`(Z) F(Z)`
$\{$`F` : `Type→Type`$\}\forall$`(Z) H(F(Z))`
$\{$`G` : `Type` $\times$ `Type→Type`$\}\forall$`(Z)` $\forall$`(W) G(W, Z)`

(where `H` is a pattern variable of arity 1 introduced by an enclosing `typecase`). But we do not allow:

$\{$`F` : `Type→Type`$\}$`F(Int)`
$\{$`F` : `Type→Type`, `H` : `Type→Type`$\}\forall$`(Z) F(H(Z))`
$\{$`G` : `Type` $\times$ `Type→Type`$\}\forall$`(Z) G(Z, Z)`

because, according to requirement (2), F must be applied to a type variable rather than to Int or to H(Z), and G must be applied to two distinct type variables rather than to Z twice.

At run time, we must solve the problem of matching a tag against a pattern. The cases where the pattern's outermost construct is a type variable, $\rightarrow$, or $\forall$ are all evident. Hence, the problem of matching a tag against a pattern is reduced to matching subproblems of the form $\mathtt{F}(\mathtt{X}_1, \ldots, \mathtt{X}_k) = \mathtt{A}$, where F is a pattern variable introduced in the pattern, $\mathtt{X}_1, \ldots, \mathtt{X}_k$ are distinct type variables, and A is a subexpression of the tag. We require:

3. the free type variables of a tag subexpression matching a pattern variable are exactly the arguments to the pattern variable

or, in this instance, $\mathtt{X}_1, \ldots, \mathtt{X}_k$ are the free type variables of A. If this holds, we take $\mathtt{F} = \Lambda(\mathtt{X}_1) \ldots \Lambda(\mathtt{X}_k)\ \mathtt{A}$ as the solution for the subproblem. This solution is evidently unique, and thus definiteness is guaranteed. Moreover, requirement (3) implies that the function $\Lambda(\mathtt{X}_1) \ldots \Lambda(\mathtt{X}_k)\ \mathtt{A}$ has the desirable properties of being closed and of using all of its arguments.

For example, we obtain a success with

Tag:	$\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{Z}\rightarrow(\mathtt{Z}\rightarrow\mathtt{Z})$
Pattern:	$\{\mathtt{F} : \mathtt{Type}\rightarrow\mathtt{Type}\}\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{F}(\mathtt{Z})$
Result:	$\{\mathtt{F} = \Lambda(\mathtt{X})\ \mathtt{X}\rightarrow(\mathtt{X}\rightarrow\mathtt{X})\}$

but failures with

Tag:	$\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{Z}\rightarrow(\mathtt{W}\rightarrow\mathtt{W})$
Pattern:	$\{\mathtt{F} : \mathtt{Type}\rightarrow\mathtt{Type}\}\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{F}(\mathtt{Z})$
Result:	*failure*

and

Tag:	$\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{Int}\rightarrow(\mathtt{Int}\rightarrow\mathtt{Int})$
Pattern:	$\{\mathtt{F} : \mathtt{Type}\rightarrow\mathtt{Type}\}\forall(\mathtt{Z})\ \forall(\mathtt{W})\ \mathtt{F}(\mathtt{Z})$
Result:	*failure*

In the second example, W is a free variable of the tag subexpression $\mathtt{Z}\rightarrow(\mathtt{W}\rightarrow\mathtt{W})$ but not an argument in F(Z). Requirement (3) prevents the escape of W from the scope of its binder. In the third example, Z is an argument in F(Z) but does not occur in the tag subexpression $\mathtt{Int}\rightarrow(\mathtt{Int}\rightarrow\mathtt{Int})$. Requirement (3) guarantees that later, successful matches instantiate all new pattern variables: if F were allowed not to use its argument Z, a later pattern

$$\{\mathtt{V} : \mathtt{Type}\}\mathtt{F}(\mathtt{V})$$

might succeed with V undetermined.

The cost of our requirements may be some inconvenience. We adopt them for the sake of soundness and simplicity.

4 Implicit Polymorphism

In this section we investigate dynamics in an implicitly typed language, the core language of ML.

The general treatment of dynamics for explicitly typed languages can be applied directly to ML, thus providing the language with explicitly tagged dynamics. In this extension of ML, types can still be inferred for all constructs except dynamics; the user needs to provide type information only when creating or inspecting dynamics. For instance, consider the following program:

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

To verify its correctness, we first infer the type scheme `∀(Z)(Z→Z)→(Z→Z)` 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 that it is more general than the required tag `∀(Z)(Z→Z)→(Z→Z)`. Similarly, when a `typecase` succeeds, the extracted value is given the type scheme of its tag as if it had been let-bound. That is, an instance of the value can be used with different instances of the tag as in:

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

where `succ` is the successor function on integers, and `not` is the negation function on booleans.

This solution is adequate. However, it seems to go against the spirit of ML in several respects, as we discuss next.

4.1 Implicit tagging

The solution just sketched requires explicit tags in `dynamic` expressions. Since the ML typechecker can infer most general types for expressions, one would expect it to tag dynamics with their principal types. For instance, the user should be able to write:

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

and expect that the dynamic will be tagged with `∀(Z)(Z→Z)→(Z→Z)`.

If types are not to be passed as arguments to polymorphic functions at run time, the tags of dynamics must be closed (see section 3.3). This restriction creates difficulties for the implicit-tagging approach: a program like

```
λ(x) dynamic(x)
```

will fail to have a principal type. It will therefore simplify matters to assume that explicit tags are given in `dynamic` expressions. An alternative is discussed in [2].

4.2 Tag instantiation and tuple variables

The tag of `apply`:

$$\texttt{apply = dynamic(}\lambda\texttt{(f) }\lambda\texttt{(x) f x:}\forall\texttt{(X,Y)(X}\to\texttt{Y)}\to\texttt{(X}\to\texttt{Y))}$$

is equivalent to $\forall\texttt{(Y,X)(X}\to\texttt{Y)}\to\texttt{(X}\to\texttt{Y)}$, and an ML programmer would probably view the order of quantifiers as unimportant. In addition, the tag is more general than the pattern in the function `foo`; hence an ML programmer would probably expect the tag and the pattern to match when `apply` is passed to `foo`. In general, it seems reasonable to ignore the order of quantifiers in a tag, and to let a tag match a pattern if any instance of the tag does. This principle is called *tag instantiation*.

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

- Second-order pattern variables may depend on universal variables, as in the pattern $\{\texttt{F}\}\ \texttt{(f:}\forall\texttt{(Z)F(Z)}\to\texttt{Z)}$.
- 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 $\forall\texttt{(Z)(Z}\times\texttt{Z)}\to\texttt{Z}$ matches the previous pattern; so should the tag $\forall\texttt{(X,Y)(X}\times\texttt{Y)}\to\texttt{X}$. But `F` is not supposed to depend on two variables.

Because of tag instantiation, polymorphic pattern variables may always depend on more variables than the ones explicitly mentioned. We deal with this possibility by introducing tuple variables, which stand for tuples of variables. The tuple variable in a pattern will be dynamically instantiated to the tuple of all variables of the tag not matched by other variables of the pattern. For example, using a tuple variable `U`, the pattern $\{\texttt{F}\}\ (\texttt{f} : \forall(\texttt{Z})\ \texttt{F}(\texttt{Z})\to\texttt{Z})$ should be written $\{\texttt{F}\}\ (\texttt{f} : \forall(\texttt{U},\texttt{Z})\ \texttt{F}(\texttt{U},\texttt{Z})\to\texttt{Z})$.

Tuple variables bound in different patterns may be instantiated to tuples with different numbers of variables. Because of such size considerations, it is not always possible to use a tuple variable as argument to an operator, since this operator may expect an argument of different size. We introduce a simple system of arities in order to guarantee that type expressions are well formed. Formally, our example pattern should now be written:

$$\{\pi : \texttt{Tuple}, \texttt{F} : \pi \to \texttt{Type} \to \texttt{Type}\}\ \ (\texttt{f} : \forall(\texttt{U} : \pi, \texttt{Z} : \texttt{Type})\ \texttt{F}(\texttt{U},\texttt{Z})\to\texttt{Z})$$

The arity variable π is to be bound at run time to the size of the tuple assigned to the tuple variable `U`. However, it is not necessary to write all the arities in programs since a typechecker can easily infer them.

In the following examples, we write $\texttt{U}_\texttt{i}$ for the `i`-th component of tuple variable `U`.

Tag:	$\forall(\texttt{Z})\ (\texttt{Z}\times\texttt{Z})\to\texttt{Z}$
Pattern:	$\{\pi, \texttt{F}\}\ \ (\texttt{f} : \forall(\texttt{U} : \pi, \texttt{Z})\ \texttt{F}(\texttt{U},\texttt{Z})\to\texttt{Z})$
Result:	$\{\pi = \texttt{0},\ \texttt{F} = \Lambda(\texttt{U},\texttt{Z})\ \texttt{Z}\times\texttt{Z}\}$

$$
\begin{array}{lll}
K ::= & \pi & \text{arity variables} \\
 & \mid \texttt{Type} & \text{sort of types} \\
 & & \\
J ::= & K & \\
 & \mid K \times \texttt{Type}^n \to K & \\
 & & \\
T ::= & Z & \text{type variables} \\
 & \mid F & \text{first-order pattern variables} \\
 & \mid \texttt{Dynamic} & \\
 & \mid T \to T & \\
 & \mid F(T^n) \quad (n > 0) & \text{application of a pattern operator} \\
 & & \\
S ::= & \forall (Z_1 : K_1, \ldots, Z_n : K_n)\ T & \text{type schemes} \\
 & & \\
P ::= & \{\pi, F_1 : J_1, \ldots, F_n : J_n\}\ \ S & \text{patterns} \\
 & & \\
a ::= & x & \\
 & \mid \lambda(x)\ a & \\
 & \mid a\ \ a & \\
 & \mid \texttt{let}\ x = a\ \texttt{in}\ a & \\
 & \mid \texttt{dynamic}(a : S) & \\
 & \mid \texttt{typecase}\ \ a\ \ \texttt{of}\ \ (x : P)\ a\ \texttt{else}\ \ a &
\end{array}
$$

Figure 2: Syntax for ML with `Dynamic`

Here the tuple arity is zero, thus `F` does not depend on `U`.

Tag:	$\forall(\mathtt{X}, \mathtt{Y})\ (\mathtt{X} \times \mathtt{Y}) \to \mathtt{X}$
Pattern:	$\{\pi, \mathtt{F}\}\ \ (\mathtt{f} : \forall(\mathtt{U} : \pi, \mathtt{Z})\ \mathtt{F}(\mathtt{U}, \mathtt{Z}) \to \mathtt{Z})$
Result:	$\{\pi = \mathtt{1},\ \mathtt{F} = \Lambda(\mathtt{U}, \mathtt{Z})\ \mathtt{Z} \times \mathtt{U}_1\}$

Tag:	$\forall(\mathtt{X}, \mathtt{Y})\ (\mathtt{X} \times \mathtt{Y}) \to \mathtt{X}$
Pattern:	$\{\pi, \mathtt{F}, \mathtt{G}\}\ \ (\mathtt{f} : \forall(\mathtt{U} : \pi, \mathtt{Z})\ \mathtt{F}(\mathtt{U}) \to \mathtt{G}(\mathtt{U}))$
Result:	$\{\pi = \mathtt{2},\ \mathtt{F} = \Lambda(\mathtt{U})\ \mathtt{U}_1 \times \mathtt{U}_2,\ \mathtt{G} = \Lambda(\mathtt{U})\ \mathtt{U}_1\}$

4.3 A language

We briefly consider a language with explicit tagging of dynamics and with tuple variables for tag instantiation. The syntax of the language is given in Figure 2.

Typechecking for this language is similar to typechecking for ML with local type declarations and type constraints. In addition to typechecking the core language, the well-formedness of type expressions and the correct scoping of type symbols must also be checked.

The matching algorithm is slightly complicated by tag instantiation and the use of tuple variables. As in the explicit case, we stipulate that fresh pattern variables can be applied only to distinct universally quantified variables. We also require that there be at most one universal tuple variable per pattern, and that its arity be given by the unique fresh arity variable introduced in the pattern. With these restrictions, matching becomes a simple extension of first-order unification with restricted type operators; we omit the details.

4.4 Related work

The work on dynamic typing most closely related to ours is that of Leroy and Mauny [14]. Their dynamics without pattern variables have been implemented in the CAML language [22]. Our work can be seen as an extension of their system with "mixed quantification."

Rather than introduce a `typecase` statement, Leroy and Mauny merge dynamic elimination with the usual case statement of ML. Ignoring this difference, their dynamic patterns have the form $\mathcal{Q}Z$ where Z is a type and $\mathcal{Q}$ a list of existentially or universally quantified variables. For instance,

$$\forall\texttt{(X)}\exists\texttt{(F)}\forall\texttt{(Y)}\exists\texttt{(G) (v:T(X,F,Y,G))}$$

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:

$$\exists\texttt{(F)}\exists\texttt{(G)}\forall\texttt{(X)}\forall\texttt{(Y) (v:T(X,F(X),Y,G(X,Y)))}$$

The equivalent pattern for us is:

$$\{\pi,\mathtt{F},\mathtt{G}\}\ \ (\mathtt{v} : \forall(\mathtt{U} : \pi, \mathtt{X}, \mathtt{Y})\, \mathtt{T}(\mathtt{X}, \mathtt{F}(\mathtt{U}, \mathtt{X}), \mathtt{Y}, \mathtt{G}(\mathtt{U}, \mathtt{X}, \mathtt{Y})))$$

With the same approach, in fact, we can translate all their patterns. On the other hand, some of our patterns do not seem expressible in their language, for example:

$$\{\pi,\mathtt{F},\mathtt{G}\}\ \ (\mathtt{v} : \forall(\mathtt{U} : \pi, \mathtt{X}, \mathtt{Y})\, \mathtt{T}(\mathtt{X}, \mathtt{F}(\mathtt{U}, \mathtt{X}), \mathtt{Y}, \mathtt{G}(\mathtt{U}, \mathtt{Y})))$$

because the quantifiers in the prefix of their patterns are in linear order, and cannot express the "parallel" dependencies of `F` on `X` and `G` on `Y`.

Another source of differences is our use of tuple variables. These enable us to write examples like the `applyTwice` function:

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

which applies its first argument to each of the two components of its second argument and returns the pair of the results. Such examples cannot be expressed in systems with only type quantifiers.

5 Abstract Data 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 dynamic 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.

- On the other hand, 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 [17], 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 in the previous sections yields the second solution.)

To add existential types to the variant of F defined in the previous section, we extend the syntax of types and terms as in Figure 3.

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

Figure 3: Extended syntax with existential types

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

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

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

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

```
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 ∃(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 discussed in section 3.3, 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` a `as` $[Z, x]$ `in` b by replacing the representation type variable Z by the actual representation type obtained by evaluating a.
- Alternatively, we can replace Z by a fresh 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 version and for the transparent version of the representation type:

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

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

Further experience would be useful for understanding the interaction of `Dynamic` and abstract types.

6 Subtyping

In simple languages with subtyping (e.g., [4, 8]) it is natural to extend `typecase` to perform a subtype test instead of an exact match. Consider for example the expression:

```
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`$\rightarrow$`V` and the problem of subtype-matching (`Int`$\rightarrow$`Nat`)$\leq$(`V`$\rightarrow$`V`). Both `Int`$\rightarrow$`Int` and `Nat`$\rightarrow$`Nat` are instances of `V`$\rightarrow$`V` and supertypes of `Int`$\rightarrow$`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

$$
\begin{array}{rlll}
K ::= & \texttt{Type} & & \text{the kind of types} \\
| & \texttt{Power}(K)(T) & & \text{the kind of subtypes of } T \\
\\
J ::= & K & & \\
| & K^n \rightarrow K & (n > 0) & \\
| & \texttt{Power}(K^n \rightarrow K)(F) & &
\end{array}
$$

Figure 4: Extended syntax with subtyping

for `V`. This can happen, for example, in a system with bounded quantifiers [7, 9], and in systems where the collection of base types does not form an upper semilattice. Linear patterns (where each pattern variable occurs at most once) avoid these problems, but we find linearity too restrictive.

Therefore, we take an approach different from that found in simple languages with subtyping. Our approach 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 [6]. In order to incorporate also the higher-order pattern variables, we resort to power-kinds [5].

The kind structure of section 3.2 is extended in Figure 4, where it is assumed that $T : K$ and $F : K^n \rightarrow K$. Informally, the kind $\texttt{Power(Type)}(T)$ is the collection of all the subtypes of T, and similarly the kind $\texttt{Power}(K_1 \times \ldots K_n \rightarrow K)(F)$ is the collection of all the operators of kind $K_1 \times \ldots K_n \rightarrow K$ that are pointwise in the subtype relation with F. Subtyping ($\leq$) is not a primitive notion in the syntax, but it is defined by interpreting:

$T \leq T' : K$ as $T : \texttt{Power}(K)(T')$, where $T, T' : K$
$F \leq G : (K_1 \times \ldots \times K_n \rightarrow K)$ as $F(Q_1, \ldots, Q_n) \leq G(Q_1, \ldots, Q_n) : K$,
for all $Q_1 : K_1, \ldots, Q_n : K_n$, where $F, G : (K_1 \times \ldots \times K_n \rightarrow K)$

The axiomatization of $\texttt{Power}(K)(T)$ [5] is designed to induce the expected subtyping rules. For example, $T : \texttt{Power}(T)$ says that $T \leq T$.

Because of power-kinds, we can now write patterns such as:

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

Each branch guard is used in typechecking the corresponding branch body. The shape of branch guards is $\{F_1 : K_1, \ldots, F_n : K_n\}(x : P)$ where each F_i may occur in the K_j with $j > i$ and in P. This shape fits within the normal format of typing environments, and hence it introduces no new difficulties for static typechecking.

Next we consider the dynamic semantics of **typecase** in the 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. In the example above, suppose that the tag of **dx** is (`Nat` $\times$ `Int`) $\times$ `Int`; then we have the instantiations `W` = `Nat` $\times$ `Int` and `V` = `Int`. When the matching is completed, we successfully check that `W`$\leq$(`V` $\times$ `V`).

Some examples should 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) ...
else ...
```

In this example, the tag of **dx** can be any subtype of **Int**.

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≤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′≤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).

7 Conclusion

The extension of statically-typed languages with dynamic types is rather complicated. Perhaps we have been overly ambitious. We have tried to allow as much flexibility as possible, at the cost of facing difficult matching problems. And perhaps we have not been ambitious enough. In particular, we have not provided mechanisms for dealing with multiple matches at run time, in order not to complicate the language designs or their implementations. We have also ignored the possibility of adding dynamic types to F_3, or to F_ω [10]. Only more experience will reveal the most useful variants of our approach.

We have deliberately avoided semantic considerations in this paper. It seems relatively straightforward to provide precise operational semantics and then prove subject-reduction theorems for our languages. These theorems would be extensions of those established for monomorphic languages in [1], and would guarantee the soundness of evaluation for the languages. It is an entirely different matter to define denotational semantics. In particular, open tags clearly allow the expression of a rich class of non-parametric functions (which manipulate types at run time), and these do not exist in many of the usual models for polymorphic languages.

Acknowledgements

François Bourdoncle, Richard Connor, and an anonymous referee made many useful suggestions on the presentation of this paper. Cynthia Hibbard provided editorial help.

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] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. In Peter Lee, editor, *ACM Sigplan Workshop on ML and its Applications*, pages 92–103. Technical Report CMU-CS-93-105, School of Computer Science, Carnegie Mellon University, 1992.

[3] 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.

[4] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, New York, 1986. Number 242 in Lecture Notes in Computer Science.

[5] 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, 1988.

[6] 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, New York, 1991. Springer-Verlag.

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

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

[9] 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.

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

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

[12] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 1993. Special Issue on European Symposium on Programming 1992 (to appear).

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

[14] Xavier Leroy and Michel Mauny. Dynamics in ML. In John Hughes, editor, *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, 1991.

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

[16] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[17] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

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

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

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

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

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

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