A modern optimizing compiler for a dynamically typed programming language: Standard ML of New Jersey (preliminary report)

Ian Zerny `<ian@zerny.dk>`

April 1, 2012

Abstract

We present Dynamic ML: a dynamically typed language in the ML family. We show how Dynamic ML provides modern programming-language features, such as type reflection and implicit coercions. In addition, we show how the existing optimizing compiler infrastructure of Standard ML of New Jersey outperforms that of other dynamically typed programming languages currently in wide-spread use.

Contents

1 Introduction

Standard ML [2] is an aging language by now. It appears that most of its development has stagnated. Meanwhile, development has continued for other programming languages and among their features we can find many that Standard ML lacks: aspects, contracts, duck typing, dynamic scope, dynamic typing, implicit coercions, lvalues and real variables defined by assignment, multiple inheritance, objects, runtime code evaluation, and type reflection, just to name a few. These are all notable features provided by 2.0 programming languages.

In this work, we take a first step towards modernizing ML and consider dynamic types for an ML-like language. We follow Harper's vision of a dynamically typed language as a unityped language [1] and extend Standard ML with such a unityped embedded language. We assume the reader to be mildly familiar with functional languages (such as Standard ML) and with dynamically typed languages (such as Python or JavaScript). The entire development can be found on the author's website.[1]

2 Dynamic ML

We extend the syntax of Standard ML with constructs for dynamically typed features. The extension is defined by the following BNF:

$$\begin{array}{rcll}
\texttt{decl} \ni d & ::= & \mathsf{FUN}\ f[x_1, ..., x_n] = e\ \mathsf{NUF} & \text{(recursive function definitions)} \\
\\
\texttt{exp} \ni e & ::= & \mathsf{IF}\ e\ \mathsf{THEN}\ e\ \mathsf{ELSE}\ e & \text{(conditionals)} \\
& | & \mathsf{FN}\ [x_1, ..., x_n]\ \texttt{=>}\ e\ \mathsf{NF} & \text{(anonymous functions)} \\
& | & e\$[e_1, ..., e_n] & \text{(application)} \\
& | & \mathsf{VOID} & \text{(literal onething)} \\
& | & \mathsf{TRUE}\ |\ \mathsf{FALSE} & \text{(literal booleans)} \\
& | & \mathsf{NUM}\ n & \text{(literal numbers)} \\
& | & \mathsf{STR}\ \text{"..."} & \text{(literal strings)} \\
& | & \mathsf{LIST}\ [e_1, ..., e_n] & \text{(literal lists)} \\
& | & \mathsf{as}\mathit{TYPE}\ e & \text{(type casts)} \\
& | & \mathsf{is}\mathit{TYPE}\ e & \text{(type predicates)} \\
& | & \mathsf{PRINT}\ e\ |\ \mathsf{PRINTLN}\ e & \text{(printing)}
\end{array}$$

where *TYPE* is one of the types: `VOID`, `BOOL`, `NUM`, `STR`, `LIST`, or `CLO`. Since Dynamic ML is an extension of Standard ML, `decl` and `exp` are super-sets of Standard ML declarations and expressions respectively. A program is then just a sequence of declarations (or a module or any other valid top-level declaration from Standard ML).

[1] `http://www.zerny.dk/dynamic-ml.html`

Equality. The primitive equality is the straightforward extension of structural equality in Standard ML. Two values are equal if they are of the same dynamic type and the raw data is structurally equal. For example:

```
val _ = (PRINTLN (NUM 1 == NUM 0);
         PRINTLN (NUM 0 == STR "0");
         PRINTLN (STR "0" == STR "0"));
```

will print `false`, `false`, and `true`.

Of course, this is a very simpleminded notion of equality, but luckily we can extend it to more powerful equalities if we wat. We illustrate this with a JavaScript-style semantics described in Section 5.

Type casting. Often we will need to pass the raw data of a dynamically typed value to some typed ML code. In such a case, we need explicit casts to obtain the raw value. These are done with the `as`*TYPE* type-cast expressions. For example, say we want the size of a string. We can use the Standard ML `String.size` for this:

```
FUN size[s] =
  NUM (Num.fromInt (String.size (asSTR s)))
NUF
val s = STR "Hello World!"
val n = size$[s]
```

which, when loaded, will give us the output:

```
val size = CLO fn : DML.t
val s = STR "Hello World!" : DML.t
val n = NUM 12 : DML.t
```

For the most part, such type casts will be implicit in the dynamically typed code, as can be seen here from the use of `size`. Should a type cast be applied to a dynamically typed value of a different type a `TYPE_CAST_ERROR` is raised.

Type reflection. Now that we have types at runtime we can inspect them! This is a hallmark feature of a solid dynamically typed language and Dynamic ML provides ample support for it. If we just need to check for one type we can use the `is`*TYPE* predicates. For example:

```
FUN foo[x] =
  IF isNUM x
  THEN x + NUM 42
  ELSE x
NUF
```

In other cases, we want have behaviors for several types. Here we can use the built in case analysis of Standard ML. For example, lets define a more reusable size function typical found in dynamically typed languages:

```
FUN size[x] =
  (case x
    of STR s  => NUM (Num.fromInt (String.size s))
     | LIST l => NUM (Num.fromInt (List.length l))
     | _ => VOID)
NUF
val n1 = size$[LIST [NUM 1, NUM 2]]
val n2 = size$[STR "Hello World!"]
val n3 = size$[NUM 10]
```

which yields:

```
val n1 = NUM 2 : DML.t
val n2 = NUM 12 : DML.t
val n3 = VOID : DML.t
```

In the above, we let `VOID` be the result for non-sizable values. This use of case analysis is kind of neat since we do not need to put in the `asSTR` and `asLIST` type casts in the case analysis.

3 Tools

Any real programming language comes with tools. Currently we provide a compiler (the `dmlc` program), a runtime system (the Standard ML of New Jersey system), and an editor (the dml-mode for Emacs).

3.1 The compiler: sed

The compiler for Dynamic ML is a sed-script that desugars the Dynamic ML constructs into Standard ML. This script is found in the file `dmlc`. To execute a program, simply desugar it and then run the output with Standard ML of New Jersey in the working directory containing the Dynamic ML implementation file `dml.sml`:

```
$ ./dmlc hello.dml
$ sml hello.dml.sml
...
val it = () : unit
'Hello World!'
-
```

3.2 The editor: dml-mode

The dml-mode for Emacs provides a nicer interface when working with Dynamic ML. It provides syntax highlighting and interacting with the underlying Standard ML process (which will be executing our Dynamic ML code). To load dml-mode in Emacs, hit `M-x load-file RET`, then input the path to `dml-mode/dml-mode-startup.el`. Now, opening a `.dml` file will start the *DML* mode. To load code from a buffer, simply hit `C-c C-b`. On the first go it will ask for the Standard ML process to use (`sml` should work if you have it installed). After that the code within the buffer will be compiled and then loaded and executed by Standard ML of New Jersey.

3.3 The runtime: Standard ML of New Jersey

Dynamic ML is executed by embedding it into Standard ML. This embedding defines the dynamic type as a one big recursive sum type: `DML.t`. Each type of dynamic runtime value is thus a summand in the dynamic type `DML.t`. The constructs used in the image of the embedding are contained within the Standard ML structure `DML` found in `dml.sml`.

4 Benchmarks

Having defined Dynamic ML and its execution environment we now benchmark it against Python, a popular and dynamically typed programming language.

4.1 The Fibonacci function

Consider the quintessential benchmark program: the recursively defined Fibonacci function. Here is its definition in Dynamic ML:

```
FUN fib[x] =
  IF x == NUM 0 THEN x
  ELSE IF x == NUM 1 THEN x
  ELSE fib$[x - NUM 1] + fib$[x - NUM 2]
NUF
```

and its counterpart in Python:

```
def fib(x):
    if x == 0: return x
    elif x == 1: return x
    else: return fib(x - 1) + fib(x - 2)
```

Timing these two functions applied to 35 yields a wall time of 11020 milliseconds for Dynamic ML and 20272 milliseconds for Python. That is 1.84 times slower for Python compared to Dynamic ML. Despite being defined for a functional language, the Standard ML of New Jersey implementation demonstrates its superiority as an optimizing compiler and runtime for a dynamically typed programming language.

4.2 The faster Fibonacci function

Somewhat surprisingly, we can do better yet! We can step-wise optimize our use of type reflection to improve performance of the Fibonacci function.

First, we create a "primitive" (i.e., non-dynamically typed) function inside `fib` that knows we only ever give it one argument:

```
val fib1 =
  FN [x] =>
    let fun go x =
            IF x == NUM 0 THEN x
            ELSE IF x == NUM 1 THEN x
            ELSE go (x - NUM 1) + go (x - NUM 2)
    in go x
    end
  NF
```

Here `fib1 35` runs in 6929 milliseconds.

Second, we call the inner function with the raw number data:

```
val fib2 =
  FN [x] =>
    let fun go x =
            if x = 0 then NUM x
            else if x = 1 then NUM x
            else go (Num.- (x,1)) + go (Num.-(x,2))
    in go (asNUM x)
    end
  NF
```

Here we have to be careful not to mix the raw numbers (which we do subtraction on) from the dynamically typed numbers (which we do addition on). Now `fib2 35` runs in 1487 milliseconds.

Third, we make sure we return the raw number from the inner function too:

```
val fib3 =
  FN [x] =>
    let fun go x =
            if x = 0 then x
            else if x = 1 then x
            else Num.+ (go (Num.- (x,1)),
                        go (Num.- (x,2)))
    in NUM (go (asNUM x))
    end
  NF
```

Now `fib3 35` runs in 617 milliseconds.

For each definition, Table 1 lists its total running time in milliseconds and its relative speed compared to original Dynamic ML definition. Our conclusion: not only is Dynamic ML faster than other popular dynamically typed languages, we can actually make programs even more efficient if we care to do so.

Language	RT in ms	Rel RT
Python	20272	1.84
Dynamic ML: fib	11020	1
Dynamic ML: fib1	6929	0.63
Dynamic ML: fib2	1487	0.14
Dynamic ML: fib3	617	0.06

Table 1: Running time for the Fibonacci function applied to 35

5 JavaScript-style equality

As mentioned in Section 2, the equality operator is not really what we find in most dynamically typed languages. For example, it requires the user to add annoying type conversions by hand. To avoid this we show how to define alternative interpretations of the dynamic types entirely within the language. This allows defining more user friendly operators.

In the Dynamic ML distribution, we provide a JavaScript structure, `JS`, that implements JavaScript-style conversions and operators. To redefine the Fibonacci function using JavaScript-style semantics we simply open up the JS-structure locally to the function:

```
local open JS in
  FUN fibjs[x] =
    IF x == NUM 0
    THEN x
    ELSE IF x == NUM 1
         THEN x
         ELSE fibjs$[x - NUM 1] + fibjs$[x - NUM 2]
  NUF
end
```

We have effectively extended the domain of the Fibonacci function to any dynamic value that can be interpreted as a number in a JavaScript-like way:

```
- fibjs$[STR "0"];
val it = STR "0" : t
- fibjs$[STR "1"];
val it = STR "1" : t
- fibjs$[STR "2"];
val it = NUM 1 : t
- fibjs$[STR "10"];
val it = NUM 55 : t
- fibjs$[STR ""];
val it = STR "" : t
- fibjs$[STR "-1"];
  C-c C-c
Interrupt
```

The JavaScript-like equality allows us to easily compare values of distinct types:

```
- open JS;
- STR "" == NUM 0;
val it = BOOL true : t
- NUM 0 == STR "0";
val it = BOOL true : t
```

That might look disturbing, but fret not, we cannot draw bogus conclusions such as the empty string being equal to a non-empty string:

```
- STR "" == STR "0";
val it = BOOL false : t
```

That would be nonsense.

6 Conclusion and perspectives

We have presented Dynamic ML, a dynamically typed language in the ML family. Dynamic ML shows that even though Standard ML has a static type system we don't need to use it. The language is a simple embedding into Standard ML and allows us to reuse the existing optimizing compiler and runtime infrastructure. Indeed, the execution of dynamically typed programs in Standard ML of New Jersey is faster than the execution of programs in Python.

To illustrate the ease of use and performance of Dynamic ML, we would have liked to finish with a more real-world program than the Fibonacci function. Unfortunately, our current map/reduce program fails with a `TYPE_CAST_ERROR` exception and we have not been able to debug the cause. We are now looking at building a debugger.

Our next step is to add objects. It is a bit of a stretch to call a language without an object system for a dynamically typed (or modern) language.

As illustrated in the optimizations of the Fibonacci function, a lot of time can be saved by optimizing out dynamic type casts. Future work should consider automatically optimizing these cases. We are looking at a refinement-type analysis to this effect.

We are also excited about Ohori et al.'s resent work on SML# [3], and hope that it will open new vistas for Bovik's use of thaumaturgic data bases.

Acknowledgments: This work was done while visiting CMU in the winter 2011/2012. Many thanks to Bob Harper for his lecture series on the principles of programming languages in which it became clear to the author that we can avoid the static types of Standard ML altogether.

References

[1] Robert Harper. Programming languages: Theory and practice. Working Draft v1.27. Available at `http://www.cs.cmu.edu/ rwh/plbook/`, 2012.

[2] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML, version 2. Report ECS-LFCS-88-62, University of Edinburgh, Edinburgh, Scotland, August 1988.

[3] Atsushi Ohori and Katsuhiro Ueno. Making Standard ML a practical database programming language. In Manuel M T Chakravarty, Olivier Danvy, and Zhenjiang Hu, editors, *ICFP*, pages 307–319, Tokyo, Japan, September 2011. ACM Press. Invited talk.