

Delimited overloading

Dany MASLOWSKI

`<dan_mski@hotmail.com>`

- Faculty mentor: Prof. Christophe TROESTLER
`<Christophe.Troestler@umh.ac.be>`
- License: LGPL with the standard OCaml linking exception.
- Platforms: All platforms OCaml runs on.
- Duration: 2 months.

Abstract

In scientific computing, writing mathematical formulae without appropriate syntactic sugar is often tedious and error prone. This proposal is a Camlp4 syntax-extension allowing a controlled form of overloading (in particular but not limited to arithmetic operators) that improves code readability, maintenance, and even sometimes its efficiency. This overloading will be provided out of the box for the predefined OCaml numeric types and will be easily extensible to user-defined types.

Description

Operator overloading is a controversial feature. Some people claim it is necessary in order to write readable numerical code, while others assert it makes the meaning of some expressions intractable because of the many possible symbol interpretations that are not apparent from the syntactic context.

In the ML world too the situation is divided. SML allows overloading of arithmetic and comparison operators (resolved with type annotations) but only for the basic types provided by the language. OCaml, on the other hand, does not allow any form of overloading.

The purpose of this proposal is to develop a syntax extension providing a controlled form of overloading that

- avoids the pitfall of making the code hard to understand by clearly delimiting the overloaded expression and making clear at the syntax level how the overloading is to be resolved;
- is extensible to user defined types and operations;

- does not incur any runtime cost and actually, as we shall see, may even improve the efficiency of the code.

When overloading is evoked, one certainly thinks first about the usual arithmetic and comparisons operators. For example, one wants to be able to write, say,

$$\text{Big_int} . (3 * x * x + 5 * x + 7 = 0)$$

instead of the more cumbersome

```
open Big_int
eq_big_int
  (add_big_int(mult_int_big_int 3 (mult_big_int x x))
   (add_int_big_int 7 (mult_int_big_int 5 x)))
zero_big_int
```

We will implement this kind of overloading for all OCaml number types such as `float`, `Int32`, `Int64`, `Big_int`, `Ratio`, `Num`,... Moreover it will easily be extensible to other modules (such as `Gmp`). A convenience extension will be provided so that, for any module `X`, it is easy to enable the transformation of expressions like `X . (3x+x*y)` into `X.add (X.mul (X.of_int 3) x) (X.mul x y)`. Of course, the user will also be able to map the usual arithmetic and comparison operators or new operators he wants to define to the functions of his choice. As a by-product, one will therefore allow the declaration of infixes with user-defined precedence and association rules. It will also be possible to overload function names (which can be convenient for e.g. `min`, `max`, `succ`, `mod`,...).

Sometimes however the transformations shown above are not flexible enough. That is why, at its heart, this syntax extension will allow to associate to any module name `X` a function $f_X : \text{expr} \rightarrow \text{expr}$ which will be used to transform an expression of the type `X . (e)` into $f_X(e)$. This will be used to implement a convenient and efficient way of writing complex number computations. For example, the expression

$$\text{Complex} . (3 + 2I + 4 * x + y)$$

will be transformed into something like

```
{Complex.re = 3. +. 4. *. x.Complex.re +. y.Complex.re;
  im = 2. +. 4. *. x.Complex.im +. y.Complex.im }
```

(Notice that this code is faster than if one had used `Complex.add`,...) This could also be used for many other powerful transformations. For example the symbolic

polynomial $p = q(1 + x + xy)$, where q is a variable declared earlier in the program, could be written

```
let p = Poly.(q*(1 + `x + `x * `y))
```

where the back-quote is used to introduce a symbolic variable of the polynomial.¹

Time permitting, we would also like to use this feature to enable a convenient — yet efficient — way of writing linear algebra expressions. As an example of what we have in mind, we would like an expression of the type

```
Vec.(y <-+  $\alpha_1 * x_1 + \dots + \alpha_n * x_n$ )
```

(with α_i being understood as 1 if omitted) to be automatically transformed into the `Lacaml` code

```
axpy ~alpha: $\alpha_1$  ~x:x1 y;  
:  
axpy ~alpha: $\alpha_n$  ~x:xn y;
```

(Notice that no temporaries are created in contrast with the usual overloading mechanisms.)

Automatic testing of syntax extensions is a difficult task. Nonetheless, in addition to transforming a few expressions and inspecting the generated code by hand, we are thinking of implementing some kind of “behavioral testing”: generate random expressions from a few combinators and automatically make sure the code compiles with no warnings and that its execution gives the expected result.

The documentation will of course include examples on how to use the extension for predefined numeric types as well as a short tutorial on how to define one’s own rewriting functions f_x .

Deliverables (summary)

- A `Camlp4` syntax-extension allowing a delimited form of overloading coming with default implementation for OCaml numeric types and extensible to new modules.
- Good documentation including various examples and a short tutorial on how to write expression-rewriting functions.
- A form of automatic “behavioral testing”.

¹This is convenient because, in this context, polymorphic variants make little sense and, should we need them, one can easily bind them to a variable outside the expression being transformed.