Rethink OCaml Recursive Values

Alban Reynaud

1 Introduction

OCaml is a language of the ML family. One of the features of the language is the **let rec** operators. It is usually used to define recursive functions, but it can also be used to define recursive values.

For example, the following code: **rec** x = 1 :: x creates an infinite list containing only ones.

However, not all definitions can be computed. For example, definitions like: **let rec** x = x + 1 should be rejected by the compiler.

Previously, this check relied on *ad hoc* rules [1], which caused several issues. In 2016, Jeremy Yallop proposed a new check, designed as a simplified typing system [5]. One of the problem is that his system is too complex, and it is hard to reason about.

During this internship, I investigated the current system and (following the intuition of Gabriel Scherer) and rewrote it as an inference rules system. Then, I proposed a simpler system. I implemented it in the compiler and successfully managed to pass the testsuite. Also, examining Jeremy Yallop's code, Gabriel and I found a bug and sent a pull request [2].

2 The current system

2.1 Access modes

Three *access modes* are used in this system. These modes are:

Deref : the value of a variable is accessed.

Guarded : the address of a variable is either placed in a constructor, either in an expression that is lazily evaluated, either unused.

Unguarded : the address of a variable is not used in a guarded context.

For example:

- In the expression x + 1, x is accessed in the **Deref** mode because its value must be accessed before being used as a function argument.

- In the expression 1 :: x, x is accessed in the **Guarded** mode. The address of x is used in the construction of the list, be it does not require the evaluation of x.
- In the expression **fun** () −> x, x is accessed in the **Guarded** mode, because the body of this function is lazily evaluated, so it does not require the evaluation of x.
- Considere the expression **let rec** x = **let** y = x **in** ... In the definition of y, x is neither evaluater, nor placed in a constructor, nor lazily evaluated. x is considered being used in the **Unguarded** mode.

2.2 Types and Environments

In this system, types are maps that associate variables an access mode. Intuitively, the type of an expression give the use of variables in the definition of the expression.

An environment is a map that associate variables to a type.

2.3 Operations

Some operations are defined on modes:

$$\begin{array}{lcl} \mathit{guard}(\text{Unguarded}) & = & \text{Unguarded} \\ \mathit{guard}(m \neq \text{Unguarded}) & = & m \\ \mathit{inspect}(m) & = & \text{Deref} \\ \mathit{delay}(m) & = & \text{Guarded} \\ \mathit{discard}(m) & = & \mathit{guard}(m) \end{array}$$

These operations are generalized on types.

It is possible to define an total order on modes:

$$\text{Deref} > \text{Unguarded} > \text{Guarded}$$

We have $m > m'$ if the mode m is more constraining than the mode m'.

So, let $m + m'$ denote the maximal mode between m and m', with respect to this order relation. Note that this operation is called *prec* in Jeremy Yallop's code.

We can generalize this definition for types and environments:

$$\begin{array}{lll} (t + t') & : x \to & t(x) + t'(x) \\ (\Gamma + \Gamma') & : x \to & \Gamma(x) + \Gamma'(x) \end{array}$$

with:

$$\begin{array}{lll} t(x) = \text{Guarded} & \text{if} & x \notin \mathit{dom}(t) \\ \Gamma(x) = \emptyset & \text{if} & x \notin \mathit{dom}(\Gamma) \end{array}$$

2.4 Inference Rules

Originally, Jeremy Yallop just gave an implementation of this check. Gabriel Scherer had the intuition that this system could be expressed with inference rules [3, 4].

This system works as follow: conclusions have the form

$$\Gamma \vdash e : t$$

where Γ is an environment, e is an expression and t a type. The environment is the input and the type is the ouput.

For example, the rule for application is:

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : \mathit{inpect}(t_1 + t_2)}$$

References

[1] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.05*. Chapter 7.2.

[2] Alban Reynaud and Gabriel Scherer. `https://github.com/ocaml/ocaml/pull/1915`.

[3] Gabriel Scherer. `https://github.com/ocaml/ocaml/pull/556#issuecomment-329533885`.

[4] Gabriel Scherer. `https://github.com/ocaml/ocaml/pull/556#issuecomment-329750085`.

[5] Jeremy Yallop. A new check that 'let rec' bindings are well formed. `https://github.com/ocaml/ocaml/pull/556`. Accessed: 2018-07-02.