Alban Reynaud

1 Jeremy's Checker

The current OCaml's recursive-value checker has been written by Jeremy Yallop. [2].

1.1 Modes

This checker uses three *access modes* to describe the way variables are accessed in an expression.

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.

1.2 Types and Environments

Access modes are used to describe a type-system.

In this system, the type of a variable x is a map that associate every variable used in x's definition to its access mode.

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

TODO: describe operations of types and environments (guard, discard, inspect, ...)

1.3 Inference Rules

This checker can be formalized by inference rules, as pointed out by Gabriel Scherer [1].

$$\frac{}{\Gamma \vdash c : \emptyset}\ \textit{where c is a constant.} \qquad \frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash x : \emptyset}\ \textit{when } x \notin \Gamma$$

TODO: complete the rules

1.4 The Recursive Check Algorithm

When an expression of the form `let rec` x_1 `=` e_1 `and ... and` x_n `=` e_n is encountered, an envir,

2 A new system

2.1 Overview

The checker we propose use a simpler type system. Types are just access modes rather than maps from variables to modes. Consequently, an environment is a map that associate to variables a mode.

On the previous system, with a deduction of the form: $\Gamma \vdash expr : A$, the environment Γ is the input and the type A is the output.

On this new system, with a deduction of the form: $\Gamma \vdash expr : m$, the mode m is the input and the environment Γ is the output. The idea is that m represents the mode in which the expression e will be evaluated, and the environment Γ associates each free variable of e to their use in e.

2.2 Modes

As the mode **Guarded** has different meanings, the mode **Guarded** is split into three modes:

Guarded : a variable is *guarded* if its address is placed in a constructor or stored in the environment of a closure (*TODO: add an example about this subtility, or remove it*). An expression is evaluated in a guarded context if its value is going to be used in a guarded way.

Delayed : an expression is *delayed* if it is lazily evaluated. Variables contained in a delayed expression are used in a delayed mode.

Unused : a variable that is unused.

2.3 Operations on modes

TODO: describe comparison between modes, mode composition and the rule let ... in ...

2.4 Inference Rules

$$\frac{}{\emptyset \vdash c : m} \text{ where } c \text{ is a constant} \qquad \frac{}{x : m \vdash x : m}$$

$$\frac{\Gamma_1 \vdash e_1 : m[Deref] \qquad \Gamma_2 \vdash e_2 : m[Deref]}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : m}$$

$$\frac{\Gamma_1 \vdash e_1 : m[Guarded] \qquad ... \qquad \Gamma_n \vdash e_n : m[Guarded]}{\Gamma_1 + ... + \Gamma_n \vdash K(e_1, ..., e_n) : m}$$

$$\frac{\Gamma, x : m_x \vdash e : m[Delayed]}{\Gamma \vdash fun\ x \rightarrow e : m}$$

2.5 The let expressions case

The basic `let` and `let rec` rules are:

$$\frac{\Gamma_1 \vdash e_1 : m[m_x + \mathit{Guarded}] \qquad \Gamma_2, x : m_x \vdash e_2 : m}{\Gamma_1 + \Gamma_2 \vdash \mathit{let}\ x\ = e_1\ \mathit{in}\ e_2 : m}$$

$$\frac{\Gamma_1, x : \_ \vdash e_1 : m[m_x + \mathit{Guarded}] \qquad \Gamma_2, x : m_x \vdash e_2 : m}{\Gamma_1 + \Gamma_2 \vdash \mathit{let\ rec}\ x = e_1\ \mathit{in}\ e_2 : m}$$

The reason why the mode $m[m_x + \mathit{Guarded}]$ is used is that the expression `e1` must be evaluated before `e2`. Therefore, at the time of the evaluation, this expression is not delayed. Its mode is at least **Guarded**. Then, the mode in which the bound variable `x` is used influences the mode in which the variable used to compute `e1` are used. Finally, it must be composed by the surrounding mode m.

The generalization for multiple `let` bindings is straightforward:

$$\frac{\begin{array}{c}\Gamma_1, x_1 : \_, x_2 : \_ \vdash e_1 : m[m_{x_1} + \mathit{Guarded}] \\ \Gamma_2, x_1 : \_, x_2 : \_ \vdash e_2 : m[m_{x_2} + \mathit{Guarded}] \\ \Gamma, x_1 : m_{x_1}, x_2 : m_{x_2} \vdash e : m\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma \vdash \mathit{let\ rec}\ x_1 = e_1\ \mathit{and}\ x_2 = e_2\ \mathit{in}\ e}$$

2.6 Patterns and Bindings

However, this presentation of the `let` rules is partial. Let bindings using patterns, such as `let (x, y) = e`.

There are two kind of patterns to consider:

Destructive Patterns : patterns such as `(x, y)` or `Some x`, that need an inspection of the matched expression.

Non-destructive patterns : patterns such as `x` or `_` that need no inspection.

Let p be a pattern, and $\mathit{mode}(p)$ be **Deref** if the pattern p is destructive, **Guarded** otherwise. A rule for `let` expressions with patterns is:

$$\frac{\Gamma_1 \vdash p : ms = e_1\ \mathit{in}\ m \qquad \Gamma_2, \mathit{vars}(p) : ms \vdash e_2 : m}{\Gamma_1 + \Gamma_2 \vdash \mathit{let}\ p = e_1\ \mathit{in}\ e_2 : m}$$

with the new notation:

$$\frac{\Gamma \vdash e : m[\sum ms + \mathit{mode}(p)]}{\Gamma \vdash p : ms = e\ \mathit{in}\ m}$$

where ms is the list of all the modes of the variables appearing in the pattern p, and $\sum ms$ is the maximal mode of ms (**Unused** if ms is empty).

For `match`, the rule is:

$$\frac{\Gamma \vdash e : m[\sum_i (mode(p_i) + ms_i)] \qquad (\Gamma_i, vars(p_i) : ms_i \vdash e_i : m)}{\sum_i \Gamma_i + \Gamma \vdash match\ e\ with\ (p_i \leftarrow e_i)^i : m}$$

References

[1] Gabriel Scherer. `https://github.com/ocaml/ocaml/pull/556#issuecomment-329750085`. Accessed: 2018-07-02.

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