A practical mode system for recursive definitions

Alban Reynaud, Gabriel Scherer, Jeremy Yallop

December 15, 2020
Recursive definitions

\[
\text{let rec } x = e
\]

Recursive \textit{function}: \(e\) is a function abstraction.

Sometimes we want more, for example recursive \textit{records}:
Recursive definitions

\[ \text{let rec } x = e \]

Recursive function: \( e \) is a function abstraction.

Sometimes we want more, for example recursive records:

\[
\text{type } 'a \text{ doubly\_linked\_list} = \{
\begin{array}{l}
\text{elem: } 'a;
\text{mutable prev: } 'a \text{ doubly\_linked\_list};
\text{mutable next: } 'a \text{ doubly\_linked\_list};
\end{array}
\}
\]

\[
\text{let create (x : } 'a) : 'a \text{ doubly\_linked\_list} = \\
\text{let rec loop =}
\text{\quad \{ prev = loop; elem = x; next = loop; \}}
\text{in loop}
\]
Recursive definitions

let rec \( x = e \)

Recursive function: \( e \) is a function abstraction.

Sometimes we want more, for example recursive records:

\[
\text{type 'a doubly_linked_list = }
\{ \text{elem: 'a;}
\text{mutable prev: 'a doubly_linked_list;}
\text{mutable next: 'a doubly_linked_list;}
\}
\]

let create (x : 'a) : 'a doubly_linked_list =
let rec loop =
\{ prev = loop; elem = x; next = loop; \}
in loop

How can we accept this and (statically) reject nonsensical definitions?

let rec \( x = x + 1 \)
State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values
State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values

Complex syntactic description.
Hard to trust.
Did not age very well with new language features.
State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values

Complex syntactic description.
Hard to trust.
Did not age very well with new language features.

#7231: check too permissive with nested recursive bindings
#7215: Unsoundness with GADTs and let rec
#4989: Compiler rejects recursive definitions of values
#6939: Segfault with improper use of let-rec and float arrays
Our contribution

A new safety criterion for recursive definitions as a *mode system* $\Gamma \vdash t : m$.

Modes characterize the way recursive variables are used.

Mode typing rules have an operational intuition; easy to extend to new language features.

Our proposal replaced the previous criterion in OCaml 4.08 (June 2019).

Compatibility: we seem to accept the same correct programs as before.

(The goal was to stay simple, not to accept more definitions.)
A glimpse of the mode system

\[ m ::= \text{Dereference} \mid \text{Return} \mid \text{Guard} \mid \text{Delay} \mid \text{Ignore} \]

\[ x_1 : m_1, \ldots, x_n : m_n \vdash t : m \]

Two readings of the judgment \( x : m_x \vdash t : m \):

- **left-to-right**: If you have access to \( x \) at mode \( m_x \),
  then you can safely use \( t \) at mode \( m \)

- **right-to-left**: If you want to use \( t \) at mode \( m \),
  then you need access to \( x \) at mode \( m_x \).

For more (mode system, soundness proof), see the longer talk or the paper.

https://arxiv.org/abs/1811.08134