A right-to-left type system for value recursion

Alban Reynaud
ENS Lyon, France

Gabriel Scherer
INRIA, France

Jeremy Yallop
University of Cambridge, UK

1 Introduction

In OCaml recursive functions are defined using the \texttt{let rec} operator, as in the following definition of \texttt{factorial}:

\begin{verbatim}
let rec fac x = if x = 0 then 1
  else x * (fac (x - 1))
\end{verbatim}

Besides functions, \texttt{let rec} can define recursive values, such as an infinite list ones where every element is 1:

\begin{verbatim}
let rec ones = 1 :: ones
\end{verbatim}

Here \texttt{x} is used in its own definition. Computing \texttt{1 + x} requires \texttt{x} to have a known value: this definition contains a vicious circle, and any evaluation strategy would fail.

Functional languages deal with recursive values in various ways. Standard ML simply rejects all recursive definitions except function values. At the other extreme, Haskell accepts all well-typed recursive definitions, including those that lead to infinite computation. In OCaml, safe cyclic-value definitions are accepted, and they are occasionally useful.

For example, consider an interpreter for a programming language with datatypes for ASTs and for values:

\begin{verbatim}
type ast = Fun (x, t) | ...
type value = Closure (env, x, t)
\end{verbatim}

The \texttt{eval} function builds values from environments and asts:

\begin{verbatim}
let rec eval env = function
  | ... => Fun (x, t) -> Closure(env, x, t)
\end{verbatim}

Now consider adding an ast constructor \texttt{FunRec} of \texttt{var \* var \* expr} for recursive functions: \texttt{FunRec (“f”, “x”, t)} represents the recursive function \texttt{let rec f x = t in f}. Our OCaml interpreter can use value recursion to build a closure for these recursive functions, without changing the type of the \texttt{Closure} constructor: the recursive closure simply adds itself to the closure environment ((\texttt{var \* value}) list).

\begin{verbatim}
let rec eval env = function
  | ... => FunRec (f, x, t) ->
    let rec cl = Closure((f,cl)::env, x, t) in cl
\end{verbatim}

Our new check and its implementation Until recently, the static check used by OCaml to reject vicious definitions relied on a syntactic analysis, performed on an untyped intermediate language. While we believe that the check as originally defined was correct, it proved fragile and hard to extend to the interaction of new language features with recursive definitions. Over the years, bugs were found where the check was unduly lenient. In conjunction with OCaml’s efficient recursive definition compilation scheme [Hirschowitz et al. 2009], this leniency led to segmentation faults.

Seeking to address these problems, we designed and implemented a new check for recursive definition safety based on a novel static analysis, formulated as a simple type system (which we have proved sound with respect to an existing operational semantics [Nordlander et al. 2008]), and implemented as part of OCaml’s type-checking phase. Our check was merged into the OCaml distribution in August 2018.

Moving the check from the middle end to the type checker re-stores the desirable property that \textit{compilation of well-typed programs does not go wrong}. This property is convenient for tools that reuse OCaml’s type-checker without performing compilation, such as MetaOCaml [Kiselyov 2014] (which type-checks quoted code) and Merlin [Bour et al. 2018] (which type-checks code during editing). Furthermore, some aspects of the check have delicate interactions with types, and so cannot be performed on an untyped IR (§4).

Our analysis We looked at reusing existing inference systems, but they do not appear to suit our analysis: they have a finer-grained handling of functions and functions than we need, but coarser-grained handling of cyclic data, and most do not propose effective inference algorithms. In return for a coarser analysis, our system is noticeably simpler; furthermore, it scales cleanly to the full OCaml language.

A key aspect of our approach is the idea of right-to-left (type to environment) algorithmic interpretation, which reduces complexity compared to a presentation designed for a left-to-right reading. It is novel in this space and could inspire other inference rules designers.

2 Static and dynamic semantics

Syntax Figure 1 introduces a minimal subset of ML with the interesting ingredients of OCaml’s recursive value definitions: a multi-ary \texttt{let rec} binding \texttt{let rec (x_1 = t_1) \ldots in u}, functions (\texttt{\lambda} abstractions) \texttt{\lambda x. t} and applications \texttt{t u}, datatype constructors \texttt{K (t_1, t_2, \ldots)} and shallow pattern-matching \texttt{match t with (K_i x_{i,1} \ldots x_{i,l_i}) \rightarrow u_i}}.

Other ML constructs (non-recursive \texttt{let}, tuples, conditionals, etc.) can be desugared into this core. In fact, the full inference rules for OCaml (and our check) exactly correspond to the rules (and check) derived from this desugaring.

Since ML’s types are largely orthogonal to our analysis, we present the check using an untyped fragment. (In the full OCaml language, there are some interactions with types — in particular, with GADTs — see §4.) Although we ignore types, we do assume that terms are well-scaled — n.b. in \texttt{let rec (x_1 = v_1) \ldots in u}, the \texttt{(x_i)} are in scope of \texttt{u} but also of all the \texttt{v_i}.

Access modes For each recursive binding \texttt{x = e}, our analysis assigns an access \texttt{mode m} representing the way that \texttt{x} is accessed during evaluation of \texttt{e}.

Figure 2 defines the modes, their order structure, and the mode composition operations. The modes are as follows:

\begin{itemize}
  \item \texttt{Ignore} : an expression is entirely unused during the evaluation of the program. This is the mode of a variable in an expression in which it does not occur.
\end{itemize}

...
Terms $\exists t, u ::= x, y, z$
| let rec $b$ in $u$
| $\lambda x, t | t u$
| $K (t_i)^j | \text{match } t \text{ with } h$

Bindings $\exists b ::= (x_i = t_i)^j$

 Handlers $\exists h ::= (p_1 \rightarrow t_i)^j$

Patterns $\exists p, q ::= K(x_i)^j$

Figure 1. Core language syntax

$$
\begin{align*}
(\Gamma, (x_j : m_{i,j})^{\epsilon_1} t_i : \text{Return}^{\epsilon_1} (m_{i,j} \leq \text{Guard})^{\epsilon_1} \\
(\Gamma'_i = \Gamma + \sum (m_{i,j})^{\epsilon_1} (t_j)^j)^j \\
(x_i : (\Gamma'_i)^{\epsilon_1} \text{ rec } (x_i = t_i)^j)^{\epsilon_1}
\end{align*}
$$

Figure 2. Access modes and operations

$$
\begin{align*}
\Gamma \vdash t : m & \quad m > m' \\
\Gamma, x : m_x \vdash t : m [\text{Delay}] \\
\Gamma \vdash \lambda x. t : m
\end{align*}
$$

Figure 3. Mode inference rules (abridged)

Modes: \( \text{Ignore} \times \text{Delay} \times \text{Guard} \times \text{Return} \times \text{Deref} \)

Mode composition:

<table>
<thead>
<tr>
<th>$m [m']$</th>
<th>\text{Ignore}</th>
<th>\text{Delay}</th>
<th>\text{Guard}</th>
<th>\text{Return}</th>
<th>\text{Deref}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Ignore}</td>
<td>\text{Ignore}</td>
<td>\text{Ignore}</td>
<td>\text{Ignore}</td>
<td>\text{Ignore}</td>
<td>\text{Ignore}</td>
</tr>
<tr>
<td>\text{Delay}</td>
<td>\text{Ignore}</td>
<td>\text{Delay}</td>
<td>\text{Delay}</td>
<td>\text{Delay}</td>
<td>\text{Deref}</td>
</tr>
<tr>
<td>\text{Guard}</td>
<td>\text{Ignore}</td>
<td>\text{Delay}</td>
<td>\text{Guard}</td>
<td>\text{Guard}</td>
<td>\text{Deref}</td>
</tr>
<tr>
<td>\text{Return}</td>
<td>\text{Ignore}</td>
<td>\text{Delay}</td>
<td>\text{Guard}</td>
<td>\text{Return}</td>
<td>\text{Deref}</td>
</tr>
<tr>
<td>\text{Deref}</td>
<td>\text{Ignore}</td>
<td>\text{Delay}</td>
<td>\text{Deref}</td>
<td>\text{Deref}</td>
<td>\text{Deref}</td>
</tr>
</tbody>
</table>

Delay : a context can be evaluated (to Weak Normal Form) without evaluating its argument. $\lambda x. \square$ is a delay context.

Guard : the context returns the value as a member of a data structure (e.g., a variant record). $K (\square)$ is a guard context. The value can safely be defined mutually-recursively with its context, as in $\text{let rec } x = K (x)$.

Return : the context returns its value without further inspection. This value cannot be defined mutually-recursively with its context, to avoid self-loops: in $\text{let rec } x = x$ and $\text{let rec } x = \text{let } y = x \text{ in } y$, the last occurrence of $x$ is in Return context.

Deref : the context inspects and uses the value in arbitrary ways. Such a value must be fully defined at the point of usage; it cannot be defined mutually-recursively with its context. $\square$ is a delay context.

The ordering $m < m'$ places less demanding, more permissive modes that do not involve dereferencing variables, below more demanding, less permissive modes.

Each mode is closely associated with particular expression contexts. For example, $t \square$ is a Deref context, since $t$ may access its argument in arbitrary ways, while $\lambda x. \square$ is a Delay context.

Mode composition corresponds to context composition: if an expression context $E[\square]$ uses its hole at mode $m$, and a second context $E'[\square]$ uses its hole at mode $m'$, then the composed context $E[E'[\square]]$ uses its hole at mode $m[m']$. Like context composition, mode composition is associative, but not commutative: Deref $[\text{Delay}]$ is Deref, but Delay $[\text{Deref}]$ is Delay.

Continuing the example above, the context $t (\lambda x. \square)$, formed by composing $t \square$ and $\lambda x. \square$, is a Deref context: the intuition is that the function $t$ may pass an argument to its input and then access the result in arbitrary ways. In contrast, the context $\lambda x. (t \square)$, formed by composing $\lambda x. \square$ and $t \square$, is a Delay context: the contents of the hole will not be touched before the abstraction is applied.

\[\text{Guard} \text{ is also used for terms whose result is discarded by the context. For example, } \square \text{ in a Guard context in } \text{let } x = \square \text{ in } u, \text{ if } x \text{ is not used in } u. \text{ Such terms cannot create self-loops, so we consider them guarded.}\]
A right-to-left type system for value recursion

\((\Gamma \vdash t : \text{Return})^{\kappa \ell}\) hold. The definitions are rejected if any \(\Gamma_j\) contains one of the mutually-defined \(y_j\) under the mode \(\text{Deref}\) or \(\text{Return}\) rather than \(\text{Guard}\) or \(\text{Delay}\).

**Subsumption** We have a subtyping/subsumption rule; for example, if we want to check \(t\) under the mode \(\text{Guard}\), it is always sound to attempt to check it under the stronger mode \(\text{Deref}\). More generally, \(m > m'\) means that \(m\) is more demanding than \(m'\), which means (in the usual subtyping sense) that it classifies fewer terms; a proof of \(\Gamma \vdash t : m\) suffices to conclude \(\Gamma \vdash t : m'\). Our algorithmic check does not use this rule; it is here for completeness.

**Abstraction and application** The rule for abstraction is discussed above. The application rule checks both function and argument in a \(\text{Deref}\) context, and merges the two resulting environments, taking the most demanding mode on each side; a variable \(y\) is dereferenced by \(u\) if it is dereferenced by either \(t\) or \(u\). The constructor rule (elided; it may be found in the full paper) is similar, but constructor parameters appear in \(\text{Guard}\) context, rather than \(\text{Deref}\).

**Recursive definitions** The rule for mutually-recursive definitions
\[
\text{let rec } b \in u \text{ is split into two parts with disjoint responsibilities. First, the binding judgment } (x_i : \Gamma_i)^{\kappa \ell} \vdash \text{rec } b \text{ computes, for each definition } x_i = e_i \text{ in a recursive binding } b, \text{ the guard } \Gamma_i \text{ of the ambient context before the binding} - \text{we detail its definition below.}
\]

Second, the \(\text{let rec } b \in u\) rule of the term judgment takes these \(\Gamma_i\) and uses them under a composition \(m'_\ell(\Gamma_i)\), to account for the actual access mode of the variables. (Here \(m(\Gamma)\) denotes the pointwise lifting of composition for each mode in \(\Gamma\).) The access mode \(m'_\ell\) is a combination of the access mode in the body \(u\) and \(\text{Guard}\), used to indicate that our eager language will compute the values now, even if they are not used in \(u\), or only under a delay.

**Binding judgment and mutual recursion** The binding judgment \((x_i : \Gamma_i)^{\kappa \ell} \vdash \text{rec } b\) is independent of the ambient context and access mode; it checks recursive bindings in isolation in the \(\text{Return}\) mode, and relates each name \(x_i\) introduced by the binding \(b\) to an environment \(\Gamma_i\) on the ambient free variables.

In the first premise, for each binding \((x_i = e_i)\) in \(b\), we check the term \(\Gamma_i\) in a context split in two parts, some usage context \(\Gamma_i\) on the ambient context around the recursive definition, and a context \((x_j : m_{i,j})^{\kappa \ell} \vdash r\) for the recursively-bound variables, where \(m_{i,j}\) is the mode of use of \(x_j\) in the definition of \(x_i\).

The second premise checks that the modes \(m_{i,j}\) are \(\leq \text{Guard}\), to ensure that these mutually-recursive definitions are valid. The third premise makes mutual-recursion safe by turning the \(\Gamma_i\) into bigger contexts \(\Gamma_i'\) taking transitive mutual dependencies into account: if a definition \(x_i = e_i\) uses the mutually-defined variable \(x_j\) under the mode \(m_{i,j}\), then we ask that the final environment \(\Gamma_i'\) for \(e_i\) contains what you need to use \(e_j\) under the mode \(m_{i,j}\), that is \(m_{i,j} \leq \Gamma_i'\). This set of equations corresponds to the fixed point of a monotone function, so it has a unique least solution.

**Note:** because the \(m_{i,j}\) must be \(\text{Guard}\), we can show that \(m_{i,j}(\Gamma_j) \leq \Gamma_j\). In particular, if we have a single recursive binding, we have \(\Gamma_i \geq m_{i,j}(\Gamma_j)\), so the third premise is equivalent to just \(\Gamma_i' \leq \Gamma_i\). The \(\Gamma_i'\) and \(\Gamma_j\) only differ for non-trivial mutual recursion.

The full paper develops meta-theoretic properties of our inference rules, such as principality.

### 3 Meta-theory: soundness

The full paper connects our inference rules to the operational semantics of Nordlander, Carlsson, and Gill [2008], with a more detailed consideration of what it means for a term to go wrong (which turns out to be quite subtle).

We define a notion of \(\text{forcing context}\) — a context that really accesses the value of its hole — and a \(\text{vicious}\) term as one with a forcing context containing a variable whose definition has not been evaluated. Then we show soundness via the following theorems:

**Lemma 3.1** (Forcing-deref). If, for a forcing context \(E_i\), \(x : m \vdash E_i[x] : \text{Return}\) is derivable, then \(m\) is \(\text{Deref}\).

**Theorem 3.2** (Vicious). \(\emptyset \vdash t\) Return never holds for \(t \in \text{Vicious}\).

**Theorem 3.3** (Subj.red.). If \(\Gamma \vdash t : m\) and \(t \rightarrow t'\) then \(\Gamma \vdash t' : m\).

**Corollary 3.4.** Return-type programs cannot go vicious.

### 4 Extension to a full language: GADTs

The combination of efficient compilation, non-uniform value representation, and features (GADTs, first-class modules) with subtle interactions between types and values introduces several challenges for checking recursive definitions in the full OCaml language. We sketch how our system naturally extends to one such challenge.

At the point where the original syntactic check took place, on an untyped IR quite late in the compiler pipeline, exhaustive single-clause matches such as \(\text{match } t \text{ with } () \rightarrow \ldots\) had been transformed into direct substitutions. With this design, programs of the following form are accepted:

\[
\text{type } t = \text{Foo} \\
\text{let rec } x = \text{match } x \text{ with } \text{Foo} \rightarrow \text{Foo}
\]

This appears innocuous, but it becomes unsound with the addition of GADTs to the language [Dolan 2016]:

\[
\text{type } (\ldots) \text{ eq } \text{Refl} : (\text{a, a}) \text{ eq} \\
\text{let all_eq (type } (a \ b) : (a, b) \text{ eq =} \\
\text{let rec (p : (a, b) eq ) = match p with Refl } \rightarrow \text{Refl in p}
\]

For the GADT \(\text{eq}\), matching against \(\text{Refl}\) is not a no-op: it brings a type equality into scope that increases the number of types that can be assigned to the program [Garrigue and Rémy 2013]. It is therefore necessary to treat matches involving GADTs as inspections to ensure that a value of the appropriate type is actually available; without that change definitions such as \(\text{all_eq}\) violate type safety.

### 5 Closing remarks

We have presented a new static analysis for recursive value declarations, designed to solve a fragility issue in the OCaml language semantics and implementation. It is less expressive than previous works that analyze function calls in a fine-grained way; in return, it remains fairly simple, despite its ability to scale to a fully-fledged programming language, and the constraint of having a direct correspondence with a simple inference algorithm.

We believe that this analysis may be of use for other functional languages, both typed and untyped. It seems likely that the techniques we have used will apply to other systems — type parameter variance, type constructor roles, and so on. Our hope in describing our system is that we will eventually see a pattern emerge for designing "things that look like type systems" in this way.

For reasons of space we refer the reader to the full paper [Reynaud et al. 2018] for a discussion of related work.
References