# 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 `let rec` operator, as in the following definition of *factorial*:

```
let rec fac x = if x = 0 then 1
                else x * (fac (x - 1))
```

Beside functions, `let rec` can define recursive values, such as an infinite list `ones` where every element is `1`:

```
let rec ones = 1 :: ones
```

Note that this "infinite" list is actually cyclic, and consists of a single cons-cell referencing itself.

However, not all recursive definitions can be computed. The following definition is justly rejected by the compiler:

```
let rec x = 1 + x
```

Here `x` is used in its own definition. Computing `1 + x` requires `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:

```
type ast = Fun of var * expr | ...
type value = Closure of env * var * expr | ...
```

The `eval` function builds values from environments and `ast`s

```
let rec eval env = function
  | ...
  | Fun (x, t) -> Closure(env, x, t)
```

Now consider adding an `ast` constructor `FunRec of var * var * expr` for recursive functions: `FunRec ("f", "x", t)` represents the recursive function `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 `Closure` constructor: the recursive closure simply adds itself to the closure environment (`(var * value) list`).

```
let rec eval env = function
  | ...
  | Fun (x, t) -> Closure(env, x, t)
  | FunRec (f, x, t) ->
    let rec cl = Closure((f,cl)::env, x, t) in cl
```

***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 restores the desirable property that *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 functors 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 `let rec` binding `let rec` $(x_i = t_i)^i$ `in` $u$, functions ($\lambda$-abstractions) $\lambda x.\, t$ and applications $t u$, datatype constructors $K\,(t_1, t_2, \dots)$ and shallow pattern-matching `match` $t$ `with` $(K_i\,(x_{i,j})^j \to u_i)^i$.

Other ML constructs (non-recursive `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-scoped — n.b. in `let rec` $(x_i = v_i)^i$ `in` $u$, the $(x_i)^i$ are in scope of $u$ but also of all the $v_i$.

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

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

- `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.

$$\begin{aligned}\text{Terms} \ni t, u ::=\ & x, y, z \\ &\mid \texttt{let rec}\ b\ \texttt{in}\ u \\ &\mid \lambda x.\, t \mid t\ u \\ &\mid K\,(t_i)^i \mid \texttt{match}\ t\ \texttt{with}\ h\end{aligned}$$

$$\begin{aligned}\text{Bindings} \ni b\ &::=\ (x_i = t_i)^i \\ \text{Handlers} \ni h\ &::=\ (p_i \to t_i)^i \\ \text{Patterns} \ni p, q &::=\ K\,(x_i)^i\end{aligned}$$

**Figure 1.** Core language syntax

Modes: Ignore ≺ Delay ≺ Guard ≺ Return ≺ Deref

Mode composition:

| $m\,[m']$ | Ignore | Delay | Guard | Return | Deref | $m$ |
|---|---|---|---|---|---|---|
| Ignore | Ignore | Ignore | Ignore | Ignore | Ignore | |
| Delay | Ignore | Delay | Delay | Delay | Deref | |
| Guard | Ignore | Delay | Guard | Guard | Deref | |
| Return | Ignore | Delay | Guard | Return | Deref | |
| Deref | Ignore | Delay | Deref | Deref | Deref | |
| $m'$ | | | | | | |

**Figure 2.** Access modes and operations

$$\frac{\Gamma \vdash t : m \qquad m \succ m'}{\Gamma \vdash t : m'} \qquad \frac{\Gamma, x : m_x \vdash t : m\,[\text{Delay}]}{\Gamma \vdash \lambda x.\, t : m} \qquad \frac{\Gamma_t \vdash t : m\,[\text{Deref}] \qquad \Gamma_u \vdash u : m\,[\text{Deref}]}{\Gamma_t + \Gamma_u \vdash t\ u : m}$$

$$\frac{(\Gamma_i, (x_j : m_{i,j})^{j \in I} \vdash t_i : \text{Return})^{i \in I} \qquad (m_{i,j} \preceq \text{Guard})^{i,j} \qquad (\Gamma'_i = \Gamma_i + \sum (m_{i,j}\left[\Gamma'_j\right])^j)^i}{(x_i : \Gamma'_i)^{i \in I} \vdash \texttt{rec}\ (x_i = t_i)^{i \in I}}$$

$$\frac{(x_i : \Gamma_i)^i \vdash \texttt{rec}\ b \qquad (m'_i)^i \stackrel{\text{def}}{=} (\max(m_i, \text{Guard}))^i \qquad \Gamma_u, (x_i : m_i)^i \vdash u : m}{\sum (m'_i\,[\Gamma_i])^i + \Gamma_u \vdash \texttt{let rec}\ b\ \texttt{in}\ u : m}$$

**Figure 3.** Mode inference rules (abridged)

- 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 or record). $K\,(\square)$ is a guard context. The value can safely be defined mutually-recursively with its context, as in `let rec` $x = K\,(x)$[1].
- Return : the context returns its value without further inspection. This value cannot be defined mutually-recursively with its context, to avoid self-loops: in `let rec` $x = x$ and `let rec` $x =$ `let` $y = x$ `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. `match` $\square$ `with` $h$ is a Deref context.

The ordering $m \prec m'$ 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 [Delay] is Deref, but Delay [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.

[1]Guard is also used for terms whose result is discarded by the context. For example, $\square$ is in a Guard context in `let` $x = \square$ `in` $u$, if $x$ is not used in $u$. Such terms cannot create self-loops, so we consider them guarded.

Finally, Ignore is the absorbing element of mode composition ($m\,[\text{Ignore}] = \text{Ignore} = \text{Ignore}\,[m]$), Return is an identity ($\text{Return}\,[m] = m = m\,[\text{Return}]$), and composition is idempotent ($m\,[m] = m$).

***A right-to-left inference system*** Figure 3 gives a representative sample of the inference rules for a judgment of form $\Gamma \vdash t : m$ for term $t$, access mode $m$ and environment $\Gamma$ that maps term variables to access modes. Modes classify terms and variables, playing the role of types in usual type systems. The example judgment $x : \text{Deref}, y : \text{Delay} \vdash (x + 1, \text{lazy}\ y) : \text{Guard}$ can be read either

- **left-to-right:** If $x$ can safely be used in Deref mode, and $y$ in Delay mode, then $(x + 1, \text{lazy}\ y)$ can safely be used at Guard.
- **right-to-left:** If a context accesses the term $(x+1, \text{lazy}\ y)$ under mode Guard, then $x$ is accessed at Deref, and $y$ at Delay.

This judgment uses access modes to classify both variables and the constraints imposed on a term by its surrounding context. If $C[\square]$ uses its hole $\square$ at the mode $m$, then any derivation for $C[t] : \text{Return}$ will contain a sub-derivation of the form $t : m$.

Mode composition features in each term rule: if we try to prove $C[t] : m'$, then the sub-derivation will check $t : m'\,[m]$, where $m'\,[m]$ is the composition of the access-mode $m$ under a surrounding access mode $m'$, and Return is neutral for composition.

Our judgment $\Gamma \vdash t : m$ can be directed into an algorithm following our right-to-left interpretation. Given a term $t$ and an mode $m$ as inputs, our algorithm computes the least demanding environment $\Gamma$ such that $\Gamma \vdash t : m$ holds.

For example, the rule for abstraction in Figure 3 has the following right-to-left reading: to compute the constraints $\Gamma$ on $\lambda x.\, t$ in a context of mode $m$, it suffices to check the body $t$ under the weaker mode $m\,[\text{Delay}]$, and remove the variable $x$ from the collected constraints – its mode does not matter. If $t$ is a variable $y$ and $m$ is Return, the resulting environment is $y : \text{Delay}$.

Given a family of mutually-recursive definitions `let rec` $(x_i = t_i)^{i \in I}$, we run our algorithm on each $t_i$ at the mode Return, and obtain a family of environments $(\Gamma_i)^{i \in I}$ such that all the judgments

$(\Gamma_i \vdash t_i : \mathrm{Return})^{i \in I}$ hold. The definitions are rejected if any $\Gamma_i$ contains one of the mutually-defined $x_j$ under the mode Deref or Return rather than Guard or Delay.

*Subsumption* We have a subtyping/subsumption rule; for example, if we want to check $t$ under the mode Guard, it is always sound to attempt to check it under the stronger mode Deref. More generally, $m \succ m'$ 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 Deref context, and merges the two resulting environments, taking the most demanding mode on each side; a variable $y$ is dereferenced by $t\ u$ 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 Guard context, rather than Deref.

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

Second, the `let rec` $b$ `in` $u$ rule of the term judgment takes these $\Gamma_i$ and uses them under a composition $m'_i\,[\Gamma_i]$, 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 $m'_i$ is a combination of the access mode in the body $u$ and 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* $(x_i : \Gamma_i)^{i \in I} \vdash \texttt{rec}\ b$ is independent of the ambient context and access mode; it checks recursive bindings in isolation in the 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 $(x_i = t_i)$ in $b$, we check the term $t_i$ in a context split in two parts, some usage context $\Gamma_i$ on the ambient context around the recursive definition, and a context $(x_j : m_{i,j})^{j \in I}$ 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 $\preceq$ 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 $m_{i,j}\left[\Gamma'_j\right]$. 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 below Guard, we can show that $m_{i,j}\left[\Gamma_j\right] \preceq \Gamma_j$. In particular, if we have a single recursive binding, we have $\Gamma_i \succeq m_{i,i}\,[\Gamma_i]$, so the third premise is equivalent to just $\Gamma'_i \stackrel{\text{def}}{=} \Gamma_i$: the $\Gamma'_i$ and $\Gamma_i$ 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 *forcing context* — a context that really accesses the value of its hole — and a *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* $E_f$, $\Gamma, x : m \vdash E_f[x] : \mathrm{Return}$ *is derivable, then* $m$ *is* Deref.

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

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

**Corollary 3.4.** Return-*typed 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 `match t with () -> ...`) had been transformed into direct substitutions. With this design, programs of the following form are accepted:

```
type t = Foo
let rec x = match x with Foo -> Foo
```

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

```
type (_, _) eq = Refl : ('a, 'a) eq
let all_eq (type a b) : (a, b) eq =
 let rec (p: (a,b) eq) = match p with Refl->Refl in p
```

For the GADT `eq`, matching against `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 `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

Frédéric Bour, Thomas Refis, and Gabriel Scherer. Merlin: a language server for OCaml (experience report). *PACMPL*, 2(ICFP):103:1–103:15, 2018.

Stephen Dolan. Unsoundness with gadts and let rec. OCaml bug report 7215, April 2016. https://caml.inria.fr/mantis/view.php?id=7215.

Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In *APLAS*, 2013.

Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. *Higher Order Symbol. Comput.*, 22(1), March 2009.

Oleg Kiselyov. The design and implementation of BER MetaOCaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014. ISBN 978-3-319-07150-3.

Johan Nordlander, Magnus Carlsson, and Andy J. Gill. Unrestricted pure call-by-value recursion. In *ML Workshop*, 2008.

Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. A right-to-left type system for mutually-recursive value definitions. *CoRR*, abs/1811.08134, 2018.