# A right-to-left type system for mutually-recursive value definitions

Alban Reynaud[1], Gabriel Scherer[2], and Jeremy Yallop[3]

[1] ENS Lyon
[2] INRIA
[3] University of Cambridge

**Abstract.** In call-by-value languages, some mutually-recursive value definitions can be safely evaluated to build recursive functions or cyclic data structures, but some definitions ('let rec x = x + 1') contain vicious circles and their evaluation fails at runtime. We propose a new static analysis to check the absence of such runtime failures.
We present a set of declarative inference rules, prove its soundness with respect to the reference source-level semantics of Nordlander, Carlsson, and Gill (2008), and show that it can be (right-to-left) directed into an algorithmic check in a surprisingly simple way.
Our implementation of this new check replaced the existing check used by the OCaml programming language, a fragile syntactic/grammatical criterion which let several subtle bugs slip through as the language kept evolving. We document some issues that arise when advanced features of a real-world functional language (exceptions in first-class modules, GADTs, etc.) interact with safety checking for recursive definitions.

## 1 Introduction

OCaml is a statically-typed functional language of the ML family. One of the features of the language is the `let rec` operator, which is usually used to define recursive functions. For example, the following code defines the factorial function:

```
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 small 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 takes an environment and an `ast` and builds a `value`

```
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 definition of `Closure`: 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 clo = Closure((f,clo)::env, x, t)
                         in clo
```

Until recently, the static check used by OCaml to reject vicious recursive definitions relied on a syntactic/grammatical description and was performed on an intermediate representation in the compiler. While we believe that the check as originally defined was correct, it proved fragile and difficult to maintain as the language evolved and new features interacted with recursive definitions. Over the years, several bugs were found where the check was unduly lenient. In conjunction with OCaml's efficient compilation scheme for recursive definitions (Hirschowitz et al., 2009), this leniency resulted in memory safety violations, and 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 implementation was merged into the OCaml distribution in August 2018.

Moving the check from the middle end to the type checker restores, for recursive value definitions, the desirable property that *well-typed programs do not* *go wrong*. This property is convenient for tools that reuse OCaml's type-checker without performing compilation, such as the multi-stage language MetaOCaml Kiselyov (2014) (which type-checks code quotations) and the Merlin language server Bour et al. (2018) (which type-checks code during editing).

The present document formally describes our analysis using a core ML language restricted to the salient features for value recursion (§3). We present inference rules (§4), study the meta-theory of the analysis, and show that it is sound with respect to the operational semantics proposed by Nordlander et al. (2008) (§5). We also discuss the challenges caused by scaling the analysis to OCaml (§6), a full-fledged functional language, in particular the delicate interactions with non-uniform value representations (§6.2), with exceptions and first-class modules (§6.3), and with Generalized Algebraic Datatypes (GADTs) (§6.4).

**Contributions** We studied related work in search of an inference system that could be used, as-is or with minor modifications, for our analysis – possibly neglecting finer-grained details of the system that we do not need. We did not find any. Existing systems, detailed in Section 7.2 (Related work), have a finer-grained handling of functions (in particular ML functors), but coarser-grained handling of cyclic data, and most do not propose effective inference algorithms.

We claim the following contributions:

– We propose a new system of inference rules that captures the needs of OCaml (or F♯) recursive value definitions, previously described by ad-hoc syntactic restrictions (§4). We implemented a checker derived from these rules, scaled up to the full OCaml language and integrated in the OCaml implementation.
– We prove the analysis sound with respect to a pre-existing source-level operational semantics: accepted recursive terms evaluate without vicious-circle failures (§5).
– Our analysis is less fine-grained on functions than existing works (thanks to a less demanding problem domain), but in exchange it is noticeably simpler.
– The idea of right-to-left computational interpretation (from type to environment) reduces complexity – a declarative presentation designed for a left-to-right reading would be more complex. It is novel in this design space and could inspire other inference rules designers.

## 2 Overview

### 2.1 Access modes

Our analysis is based on the classification of each use of a recursively-defined variable using "access modes" or "usage modes" $m$. These modes represent the degree of access needed to the value bound to the variable during evaluation of the recursive definition.

For example, in the recursive function definition

```
let rec f = fun x -> ... f ...
```

the recursive reference to `f` in the right-hand-side does not need to be evaluated to define the function value `fun x ->`... since its value will only be required later, when the function is applied. We say that, in this right-hand-side, the mode of use of the variable `f` is $\mathsf{Delay}$.

In contrast, in the vicious definition `let rec x = 1 + x` evaluation of the right-hand side involves accessing the value of `x`; we call this usage mode a $\mathsf{Dereference}$. Our static check rejects (mutually-)recursive definitions that access recursively-bound names under this mode.

Some patterns of access fall between the extremes of $\mathsf{Delay}$ and $\mathsf{Dereference}$. For example, in the cyclic datatype construction `let rec ones = 1 :: ones` the recursively-bound variable `ones` appears on the right-hand side without being placed inside a function abstraction. However, since it appears in a "guarded" position, directly beneath the value constructor `::`, evaluation only needs to access its address, not its value. We say that the mode of use of the variable `ones` is $\mathsf{Guard}$.

Finally, a variable `x` may also appear in a position where its value is not inspected, neither is it guarded beneath a constructor, as in the expression `x`, or `let y = x in y`, for example. In such cases we say that the value is "returned" directly and use the mode $\mathsf{Return}$. As with `Dereference`, recursive definitions that access variables at the mode $\mathsf{Return}$, such as `let rec x = x`, would be under-determined and are rejected.

We also use a last $\mathsf{Ignore}$ mode to classify variables that are not used at all in a term.

### 2.2 A right-to-left inference system

The central contribution of our work is a simple system of inference rules for a judgment of the form $\Gamma \vdash t : m$, where $t$ is a program term, $m$ is an access mode, and the environment $\Gamma$ maps term variables to access modes. Modes classify terms and variables, playing the role of types in usual type systems. The example judgment $x : \mathsf{Dereference}, y : \mathsf{Delay} \vdash (x + 1, \mathsf{lazy}\ y) : \mathsf{Guard}$ can be read alternatively

**left-to-right:** If we know that $x$ can safely be used in $\mathsf{Dereference}$ mode, and $y$ can safely be used in $\mathsf{Delay}$ mode, then the pair $(x + 1, \mathsf{lazy}\ y)$ can safely be used under a value constructor (in a $\mathsf{Guard}$-ed context).

**right-to-left:** If a context accesses the program fragment $(x + 1, \mathsf{lazy}\ y)$ under the mode $\mathsf{Guard}$, then this means that the variable $x$ is accessed at the mode $\mathsf{Dereference}$, and the variable $y$ at the mode $\mathsf{Delay}$.

This judgment uses access modes to classify not just variables, but also the constraints imposed on a subterm by its surrounding context. If a context $C[\square]$ uses its hole $\square$ at the mode $m$, then any derivation for $C[t] : \mathsf{Return}$ will contain a sub-derivation of the form $t : m$.

In general, we can define a notion of mode composition: 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 usage 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 inference rule for function abstractions in our system is as follows:

$$\frac{\Gamma, x : m_x \vdash t : m\,[\mathsf{Delay}]}{\Gamma \vdash \lambda x.\, t : m}$$

The right-to-left reading of the rule is as follows. To compute the constraints $\Gamma$ on $\lambda x.\, t$ in a context of mode $m$, it suffices to the check the function body $t$ under the weaker mode $m\,[\mathsf{Delay}]$, and remove the function variable $x$ from the collected constraints – its mode does not matter. If $t$ is a variable $y$ and $m$ is Return, we get the environment $y : \mathsf{Delay}$ as a result.

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

### 2.3 Issues with the previous check

Before this work, the safety criterion used by OCaml for recursive value definitions was an ad-hoc grammatical restriction, formulated essentially as a context-free grammar of accepted definitions (see its description in the reference manual). Furthermore, this syntactic check was not performed on the source program directly, but on an intermediate representation (the Lambda code) – so that it wouldn't have to take into account various surface-language forms that desugar to the same intermediate-language construct.

We list below some of the known issues with the previous check. They were solved by our work.

*PR#7231: unsoundness with nested recursive bindings* The following unsafe program was accepted by the previous check.

```
let rec r = let rec x () = r
                and y () = x ()
            in y ()
        in r "oops" (* segfault *)
```

The problem is that while the declarations of `x` and `y` are "safe" (in some sense) with respect to `r`, using `y` is not safe – it returns `r` itself. This subtlety was lost on the previous check. With the current check, `y ()` uses `r` at mode `Return`, which is stricter than Guard, so this program is rejected.

*PR#7215: unsoundness with GADTs* The following unsafe program was accepted by the previous check.

```
let is_int (type a) =
  let rec (p : (int, a) eq) = match p with Refl -> Refl in
  p
```

This program uses a recursive value declaration of a GADT value to build a type-equality between `int` and an arbitrary type `a`. Our check rejects the program because `match p with Refl -> ...` is a dereferencing use of `p`. The previous check was run on an intermediate form, after various optimizations, one of which would eliminate the single-case match away, resulting in the (unsound) program passing the check.

*PR#6939: unsoundness with float arrays*

```
let rec x = ([| x |]; 1.) in ()
```

This program defines `x` to be the floating-point value `1.` after ignoring the value of the one-element array `[| x |]`. Although the program was accepted by the previous check OCaml's non-uniform value representation makes it unsafe, and it would would fail with a segmentation fault when run, as explained in Section 6.2 (Dynamic representation checks: float arrays). Our algorithm uses typing information, which is needed to detect this case: construction of a float array is treated as a $\mathsf{Dereference}$ context for its elements.

*PR#4989: inconveniently rejected program*

```
let rec f =
  let g = fun x -> f x in
  g
```

This program, which gives a local name to an expression that accesses `f` at mode $\mathsf{Delay}$, is perfectly safe, but was rejected by the previous check. A grammar-based check lacks a form of composability that would allow the use of local bindings to give names to sub-expressions in an analysis-preserving way. This form of composability had been requested by users for a long time as a convenience feature, but the previous check could not be extended to allow it. On the contrary, proper handling of inner `let`-bindings falls out naturally from our type-system-inspired approach.

## 3 A core language of recursive definitions

*Family notation* We write $(\ldots)^{i \in I}$ for a family of objects parametrized by an index $i$ over finite set $I$, and $\emptyset$ for the empty family. Furthermore, we assume that index sets are totally ordered, so that the elements of the family are traversed in a predetermined linear order; we write $(t_{i_1})^{i_1 \in I_1}, (t_{i_2})^{i_2 \in I_2}$ for the combined family over $I_1 \uplus I_2$, with the indices in $I_1$ ordered before the indices of $I_2$. We

$$
\begin{array}{llll}
\mathsf{Terms} \ni t, u ::= & x, y, z & & \\
& \mid \texttt{let rec}\ b\ \texttt{in}\ u & \mathsf{Bindings} \ni b & ::= (x_i = t_i)^i \\
& \mid \lambda x.\, t \mid t\ u & \mathsf{Handlers} \ni h & ::= (p_i \to t_i)^i \\
& \mid K\,(t_i)^i \mid \texttt{match}\ t\ \texttt{with}\ h & \mathsf{Patterns} \ni p, q & ::= K\,(x_i)^i
\end{array}
$$

**Fig. 1.** Core language syntax

often omit the index set, writing $(\dots)^i$. Families may range over two indices (the domain is the cartesian product), for example $(t_{i,j})^{i,j}$.

Our syntax, judgments, and inference rules will often use families: for example, `let rec` $(x_i = t_i)^i$ is a mutually-recursive definition of families $(t_i)^i$ of terms bound to corresponding variables $(x_i)^i$ – assumed distinct, we follow the Barendregt convention. Sometimes a family is used where a term is expected, and the interpretation should be clear: when we say "$(\Gamma_i \vdash t_i : m_i)^i$ holds", we implicitly use a conjunctive interpretation: each of the judgments in the family holds.

### 3.1 Syntax

Figure 1 introduces a minimal subset of ML containing the interesting ingredients of OCaml's recursive values:

- A multi-ary `let rec` binding `let rec` $(x_i = t_i)^i$ `in` $u$.
- Functions ($\lambda$-abstractions) $\lambda x.\, t$ to write recursive occurrences whose evaluation is delayed.
- Datatype constructors $K\,(t_1, t_2, \dots)$ to write (safe) cyclic data structures; these stand in both for user-defined constructors and for built-in types such as lists and tuples.
- Shallow pattern-matching (`match` $t$ `with` $(K_i\,(x_{i,j})^j \to u_i)^i$), to write code that inspects values, in particular code with vicious circles.

The following common ML constructs do not need to be primitive forms, as we can desugar them into our core language. In particular, the full inference rules for OCaml (and our check) exactly correspond to the rules (and check) derived from this desugaring.

- $n$-ary tuples are a special case of constructors:
  $(t_1, t_2, \dots, t_n)$ desugars into $\texttt{Tuple}_n\,(t_i)^{i \in [1;n]}$.
- Non-recursive *let* bindings are recursive bindings with access mode $\mathsf{Ignore}$:
  `let` $x = t$ `in` $u$ desugars into `let rec` $x = t$ `in` $u$.
- Conditionals are a special case of pattern-matching:
  `if` $t$ `then` $u_1$ `else` $u_2$ desugars into `match` $t$ `with` (`True` $\to u_1$ | `False` $\to u_2$).

Besides dispensing with many constructs whose essence is captured by our minimal set, we further simplify matters by using an untyped ML fragment: we do not need to talk about ML types to express our check, or to assume that the terms we are working with are well-typed.[4] (Untyped algebraic datatypes might make you nervous, but work just fine, and were invented in that setting. A `match` form gets stuck if the head constructor of the scrutinee is not matched (with the same arity) by any clause.) However, we do assume that our terms are well-scoped – note that, in $\texttt{let rec}\ (x_i = v_i)^i\ \texttt{in}\ u$, the $(x_i)^i$ are in scope of $u$ but also of all the $v_i$.

Remark: recursive values break the assumption that structurally-decreasing recursive functions will terminate on all inputs.[5] In our experience, users of recursive values are careful to ensure termination; we are not aware of production bugs caused by cyclic data flowing into unsuspecting consumers, but writing the correct definitions can be delicate. Jeannin, Kozen, and Silva (2017) propose language extensions to make it easier to operate over such cyclic structures.

## 4 Our inference rules for recursive definitions

### 4.1 Access/usage modes

Figure 2 defines the usage/access modes that we introduced in Section 2.1, their order structure, and the mode composition operations. The modes are as follows:

Ignore is for sub-expressions that are not used at all during the evaluation of the whole program. This is the mode of a variable in an expression in which it does not occur.

Delay means that the context can be evaluated (to a weak normal-form) without evaluating its argument. $\lambda x. \Box$ is a delay context.

Guard means that the context returns the value as a member of a data structure, for example a variant constructor or record. $K\,(\Box)$ is a guard context. The value can safely be defined mutually-recursively with its context, as in $\texttt{let rec}\ x = K\,(x)$.

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

Dereference means that the context consumes, 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. $\texttt{match}\ \Box\ \texttt{with}\ h$ is a Dereference context.

[4] In more expressive settings, the structure of usage modes does depend on the structure of values, and checks need to be presented as a refinement of a ML type system. We discuss this in Section 7.2. Our modes are a degenerate case, a refinement of uni-typed ML.

[5] ML accepts general recursive types, not just inductive types with recursive occurrences in positive positions. In particular, structural recursion may not terminate even in absence of recursive values

$$\mathsf{Modes} \ni m ::= \mathsf{Ignore} \mid \mathsf{Delay} \mid \mathsf{Guard} \mid \mathsf{Return} \mid \mathsf{Dereference}$$

Mode order:
$\mathsf{Ignore} \prec \mathsf{Delay} \prec \mathsf{Guard} \prec \mathsf{Return} \prec \mathsf{Dereference}$

Mode composition rules :

$$\begin{array}{lll}
\mathsf{Ignore}\,[m] & = \mathsf{Ignore} \quad = & m\,[\mathsf{Ignore}] \\
\mathsf{Delay}\,[m \succ \mathsf{Ignore}] & = \mathsf{Delay} & \\
\mathsf{Guard}\,[\mathsf{Return}] & = \mathsf{Guard} & \\
\mathsf{Guard}\,[m \neq \mathsf{Return}] & = m & \\
\mathsf{Return}\,[m] & = m & \\
\mathsf{Dereference}\,[m \succ \mathsf{Ignore}] & = \mathsf{Dereference} &
\end{array}$$

Mode composition as a table:

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

**Fig. 2.** Access/usage modes and operations

*Remark 1 (Discarding).* The Guard mode is also used for subterms whose result is discarded by the evaluation of their context. For example, the hole $\square$ is in a Guard context in ($\texttt{let}\ x = \square\ \texttt{in}\ u$), if $x$ is never used in $u$; even if the hole value is not needed, call-by-value reduction will first evaluate it and discard it. When these subterms participate in a cyclic definition, they cannot create a self-loop, so we consider them guarded.

Our ordering $m \prec m'$ places less demanding, more permissive modes that do not involve dereferencing variables (and so permit their use in recursive definitions), below more demanding, less permissive modes.

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

Mode composition corresponds to context composition, in the sense that if an expression context $E[\square]$ uses its hole at mode $m$, and a second expression context $E'[\square]$ uses its hole at mode $m'$, then the composition of contexts $E[E'[\square]]$ uses its hole at mode $m\,[m']$. Like context composition, mode composition is associative, but not commutative: Dereference [Delay] is Dereference, but Delay [Dereference] is Delay.

Continuing the example above, the context $t\ (\lambda x.\,\square)$, formed by composing $t\ \square$ and $\lambda x.\,\square$, is a Dereference context: the intuition is that the function $t$ may pass an argument to its input and then access the result in arbitrary ways. In

$$\boxed{\text{Term judgment } \Gamma \vdash t : m}$$

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \qquad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m\,[\mathsf{Delay}]}{\Gamma \vdash \lambda x.\, t : m} \qquad \frac{\Gamma_t \vdash t : m\,[\mathsf{Dereference}] \qquad \Gamma_u \vdash u : m\,[\mathsf{Dereference}]}{\Gamma_t + \Gamma_u \vdash t\ u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m\,[\mathsf{Guard}])^i}{\sum (\Gamma_i)^i \vdash K\,(t_i)^i : m} \qquad \frac{\Gamma_t \vdash t : m\,[\mathsf{Dereference}] \qquad \Gamma_h \vdash^{\mathsf{cl}} h : m}{\Gamma_t + \Gamma_h \vdash \mathtt{match}\ t\ \mathtt{with}\ h : m}$$

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

$$\boxed{\text{Clause judgments } \Gamma \vdash^{\mathsf{cl}} h : m \text{ and } \Gamma \vdash^{\mathsf{cl}} p \to u : m}$$

$$\frac{(\Gamma_i \vdash^{\mathsf{cl}} p_i \to u_i : m)^i}{\sum (\Gamma_i)^i \vdash^{\mathsf{cl}} (p_i \to u_i)^i : m} \qquad \frac{\Gamma, (x_i : m_i)^i \vdash u : m}{\Gamma \vdash^{\mathsf{cl}} K\,(x_i)^i \to u : m}$$

$$\boxed{\text{Binding judgment } (x_i : \Gamma_i)^i \vdash \mathtt{rec}\ b}$$

$$\frac{\begin{array}{c}(\Gamma_i, (x_j : m_{i,j})^{j \in I} \vdash t_i : \mathsf{Return})^{i \in I} \qquad (m_{i,j} \preceq \mathsf{Guard})^{i,j} \\ (\Gamma_i' = \Gamma_i + \sum (m_{i,j}\,[\Gamma_j'])^j)^i\end{array}}{(x_i : \Gamma_i')^{i \in I} \vdash \mathtt{rec}\ (x_i = t_i)^{i \in I}}$$

**Fig. 3.** Mode inference rules

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.

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

### 4.2 Inference rules

*Environment notations* Our environments $\Gamma$ associate variables $x$ with modes $m$. We write $\Gamma_1, \Gamma_2$ for the union of two environments with disjoint domains, and $\Gamma_1 + \Gamma_2$ for the merge of two overlapping environments, taking the maximum mode for each variable. We sometimes use family notation for environments, writing $(\Gamma_i)^i$ to indicate the disjoint union of the members, and $\sum (\Gamma_i)^i$ for the non-disjoint merge of a family of environments.

*Inference rules* Figure 3 presents the inference rules for access/usage modes. The rules are composed into several different judgments, even though our simple core language makes it possible to merge them. In the full system for OCaml the decomposition is necessary to make the system manageable.

*Variable and subsumption rules* The variable rule is as one would expect: the usage mode of $x$ in an $m$-context is $m$. In this presentation, we let the rest of the environment $\Gamma$ be arbitrary; we could also have imposed that it map all variables to $\mathsf{Ignore}$. Our directed/algorithmic check returns the "least demanding" environment $\Gamma$ for all satisfiable judgments, so it uses $\mathsf{Ignore}$ in any case.

We have a subtyping/subsumption rule; for example, if we want to check $t$ under the mode $\mathsf{Guard}$, it is always sound to attempt to check it under the stronger mode $\mathsf{Dereference}$. Our algorithmic check will never use this rule; it is here for completeness. The direction of the comparison may seem unusual. We can coerce a $\Gamma \vdash t : m$ into $\Gamma \vdash t : m'$ when $m \succ m'$ holds, while we would expect $m \leq m'$. This comes from the fact that our right-to-left reading is opposite to the usual reading direction of type judgments, and influenced our order definition. When $m \succ m'$ holds, $m$ is *more demanding* than $m'$, which means (in the usual subtyping sense) that it classifies *fewer* terms.

*Simple rules* We have seen the $\lambda x.\, t$ rule already, in Section 2.2. Since $\lambda$ delays evaluation, checking $\lambda x.\, t$ in a usage context $m$ involves checking the body $t$ under the weaker mode $m\,[\mathsf{Delay}]$. The necessary constraints $\Gamma$ are returned, after removing the constraint over $x$[6]

The application rule checks both the function and its argument in a $\mathsf{Dereference}$ context, and merges the two resulting environments, taking the maximum (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 is similar to the application rule, except that the constructor parameters appear in $\mathsf{Guard}$ context, rather than $\mathsf{Dereference}$.

*Pattern-matching* The rule for $\texttt{match}\ t\ \texttt{with}\ h$ relies on a different *clause judgment* $\Gamma \vdash^{\mathsf{cl}} h : m$ that checks each clause in turn and merges their environments. On a single clause $K\,(x_i)^i \to u$, we check the right-hand-side expressions $u$ in the ambient mode $m$, and remove the pattern-bound variables $(x_i)^i$ from the environment.[7]

*Recursive definitions* The rule for mutually-recursive definitions $\texttt{let rec}\ b\ \texttt{in}\ u$ 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 $b$,

[6] In situations where it is desirable to have a richer mode structure to analyze function applications, as considered by some of the related work (Section 7.2), we could use the mode $m_x$ in a richer return mode $m_x \to m$.

[7] If we wanted a finer-grained analysis of usage of the sub-components of our data, we would use the sub-modes $(m_i)^i$ of the pattern variables to refine/enrich/annotate the datatype of pattern scrutinee.

the usage $\Gamma_i$ of the ambient context before the recursive 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 usage mode of the variables. (Here $m[\Gamma]$ denotes the pointwise lifting of composition for each mode in $\Gamma$.) The usage mode $m'_i$ is a combination of the usage mode in the body $u$ and $\mathsf{Guard}$, used to indicate that our call-by-value language will compute the values now, even if they are not used in $u$, or only under a delay – see Remark 1 (Access/usage modes).

*Binding judgment and mutual recursion* The *binding judgment* $(x_i : \Gamma_i)^{i \in I} \vdash$ `rec` $b$ is independent of the ambient context and usage mode; it checks recursive bindings in isolation in the $\mathsf{Return}$ 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 $\mathsf{Guard}$ or less demanding, to ensure that these mutually-recursive definitions are valid. This is the check mentioned at the end of Section 2.2 (A right-to-left inference system).

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 recursive 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 recursive equations corresponds to the fixed point of a monotone function, so in particular it has a unique least solution.

Note that because the $m_{i,j}$ must be below $\mathsf{Guard}$, we can show that $m_{i,j}[\Gamma_j] \preceq \Gamma_j$. 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 $\Gamma'_i \stackrel{\mathsf{def}}{=} \Gamma_i$: the $\Gamma'_i$ and $\Gamma_i$ only differ for non-trivial mutual recursion.

In Appendix A (Properties) we develop some direct meta-theoretic properties of our inference rules. In particular, they are principal in the sense that a unique minimal context $\Gamma$ exists for each $t : m$ – there is a unambiguous way to extract an algorithm from these rules, which we implemented in the OCaml compiler.

# 5 Meta-theory: soundness

## 5.1 Operational semantics

Figure 4 and the explanations below recall the operational semantics of Nordlander, Carlsson, and Gill (2008). (Unless explicitly noted, the content and ideas in this Subsec 5.1 come from this work.)

$$
\begin{array}{rcl}
\mathsf{Values} \ni v & ::= & \lambda x.\, t \mid K\,(w_i)^i \\
\mathsf{WeakValues} \ni w & ::= & x, y, z \mid v \\
\\
\mathsf{ValueBindings} \ni B & ::= & (x_i = v_i)^i \\
\mathsf{BindingCtx} \ni L & ::= & \square \mid \texttt{let rec}\ B\ \texttt{in}\ L \\
\\
\mathsf{EvalCtx} \ni E & ::= & \square \mid E[F] \\
\mathsf{EvalFrame} \ni F & ::= & \square\ t \ \mid\ t\ \square \\
& & \mid K\,((t_i)^i, \square, (t_j)^j) \\
& & \mid \texttt{match}\ \square\ \texttt{with}\ h \\
& & \mid \texttt{let rec}\ b, x = \square, b'\ \texttt{in}\ u \\
& & \mid \texttt{let rec}\ B\ \texttt{in}\ \square
\end{array}
$$

$$
\frac{}{L[\lambda x.\, t]\ v \to^{\mathsf{hd}} L[t[v/x]]}
$$

$$
\frac{\forall (K'\,(x'_j)^j \to u') \in h,\ K \neq K'}{\texttt{match}\ L[K\,(w_i)^i]\ \texttt{with}\ (h \mid K\,(x_i)^i \to u \mid h') \to^{\mathsf{hd}} L[u[(w_i/x_i)^i]]}
$$

$$
\frac{(x = v) \in B}{(x = v) \overset{\mathsf{frame}}{\in} \texttt{let rec}\ B\ \texttt{in}\ \square}
\qquad
\frac{(x = v) \in (b \cup b')}{(x = v) \overset{\mathsf{frame}}{\in} \texttt{let rec}\ b, y = \square, b'\ \texttt{in}\ u}
$$

$$
\frac{(x = v) \overset{\mathsf{frame}}{\in} F \vee (x = v) \overset{\mathsf{ctx}}{\in} E}{(x = v) \overset{\mathsf{ctx}}{\in} E[F]}
\qquad
\frac{t \to^{\mathsf{hd}} t'}{E[t] \to E[t']}
\qquad
\frac{(x = v) \overset{\mathsf{ctx}}{\in} E}{E[x] \to E[v]}
$$

**Fig. 4.** Operational semantics

*Weak values* As we have seen, constructors in recursive definitions can be used to construct cyclic values. For example, the definition `let rec` $x =$ `Cons` (`One` $(\emptyset), x)$ is normal for this reduction semantics. The occurrence of the variable $x$ inside the `Cons` cell corresponds to a back-reference, the cell address in a cyclic in-memory representation.

This key property is achieved by defining a class of *weak values*, noted $w$, to be either (strict) values or variables. Weak values occur in the definition of the semantics wherever a cyclic reference can be passed without having to dereference.

Several previous works (see Section 7.2 (Related work)) defined semantics where $\beta$-redexes have the form $(\lambda x.\, t)\ w$, to allow yet-unevaluated recursive definitions to be passed as function arguments. OCaml does not allow this (a function call requires a fully-evaluated argument), so our redexes are the traditional $(\lambda x.\, t)\ v$. This is a difference from Nordlander, Carlsson, and Gill (2008). On the other hand, we do allow cyclic datatype values by only requiring weak values under data constructors: besides closures $\lambda x.\, t$, the other value form is $K\,(w_i)^i$.

*Bindings in evaluation contexts* An *evaluation context* $E$ is a stack of *evaluation frames* $F$ under which evaluation may occur. Our semantics is under-constrained (for example, $t\ u$ may perform reductions on either $t$ or $u$), as OCaml has unspecified evaluation order for applications and constructors, but making it deterministic would not change much.

One common aspect of most operational semantics for `let rec`, ours included, is that `let rec` $B$ `in` $\square$ can be part of evaluation contexts, where $B$ represents a recursive "value binding", an island of recursive definitions that have all been reduced to values. This is different from traditional source-level operational semantics of `let` $x = v$ `in` $u$, which is reduced to $u[v/x]$ before going further. In `let rec` blocks this substitution reduction is not valid, since the value $v$ may refer to the name $x$, and so instead "value bindings" remain in the context, in the style of explicit substitution calculi. We call these context fragments "binding contexts" $L$.

*Head reduction* Head redexes, the sources of the head-reduction relation $t \rightarrow^{\mathsf{hd}} t'$, come from applying a $\lambda$-abstraction or from pattern-matching on a head constructor. Following ML semantics, pattern-matching is ordered: only the first matching clause is taken.

One mildly original feature of our head reduction is the use of "reduction at a distance", where binding contexts $L$ are allowed to be presented in the middle of redexes, and lifted out of the reduced term. This presentation is common in explicit-substitution calculi[8], as it gives the minimal amount of lifting of explicit substitutions required to avoid blocking reduction. In the calculus of Nordlander, Carlsson, and Gill (2008), lifting was permitted in arbitrary positions by the Merge rule.

*Reduction* Reduction $t \rightarrow t'$ may happen under any evaluation context, and is of either of two forms. The first is completely standard: any redex $H[v]$ can be reduced under an evaluation context $E$.

The second rule reduces a variable $x$ in in an evaluation context $E$ by binding lookup: it is replaced by the value of the recursive binding $B$ in the context $E$ which defines it. This uses the auxiliary definition $(x = v) \overset{\mathsf{ctx}}{\in} E$ to perform this lookup.

The lookup rule has worrying consequences for our rewriting relation: it makes it non-deterministic and non-terminating. Indeed, consider a weak value of the form $K\,(x)$ used, for example, in a pattern-matching `match` $K\,(x)$ `with` $h$. It is possible to reduce the pattern-matching immediately, or to first lookup the value of $x$ and then reduce. Furthermore, it could be the case that $x$ is precisely defined by a cyclic binding $x = K\,(x)$. Then the lookup rule would reduce to `match` $K\,(K\,(x))$ `with` $h$, and we could keep looking indefinitely. This is discussed in detail by Nordlander, Carlsson, and Gill (2008), who prove that the reduction is in fact confluent modulo unfolding. (Allowing these irritating but innocuous

[8] See for example Accattoli and Kesner (2010), which links to earlier references on the technique.

$$\begin{array}{rcl}
\mathsf{HeadFrame} \ni H & ::= & \square\, v \mid \mathtt{match}\, \square\, \mathtt{with}\, h \\
\mathsf{ForcingFrame} \ni F_{\mathsf{f}} & ::= & \square\, v \mid v\, \square \mid \mathtt{match}\, \square\, \mathtt{with}\, h \\
\mathsf{ForcingCtx} \ni E_{\mathsf{f}} & ::= & L \mid E[F_{\mathsf{f}}[L]]
\end{array}$$

$$\mathsf{Mismatch} \stackrel{\mathsf{def}}{=} \{E[H[v]] \mid H[v] \not\to^{\mathsf{hd}}\} \qquad \mathsf{Vicious} \stackrel{\mathsf{def}}{=} \{E_{\mathsf{f}}[x] \mid \nexists v, (x = v) \stackrel{\mathsf{ctx}}{\in} E_{\mathsf{f}}\}$$

**Fig. 5.** Failure terms

behaviors is a large part of what makes their semantics simpler than previous presentations.)

### 5.2 Failures

In this section, we are interested in formally defining dynamic failures. When can we say that a term is “wrong”? — in particular, when is a valid implementation of the operational semantics allowed to crash? This aspect is not discussed in detail by Nordlander, Carlsson, and Gill (2008), so we had to make our own definitions; we found it surprisingly subtle.

The first obvious sort of failure is a type mismatch between a value constructor and a value destructor: application of a non-function, pattern-matching on a function instead of a head constructor, or not having a given head constructor covered by the match clauses. These failures would be ruled out by a simple type system and exhaustivity check.

The more challenging task is defining failures that occur when trying to access a recursively-defined variable too early. The lookup reduction rule for a term $E[x]$ looks for the value of $x$ in a binding of the context $E$. This value may not exist (yet), and that may or may not represent a runtime failure.

We assume that bound names are all distinct, so there may not be several $v$ values. The only binders that we reduce under are `let rec`, so $x$ must come from one; however, it is possible that $x$ is part of a `let rec` block currently being evaluated, with an evaluation context of the form $E[\mathtt{let\ rec}\ x = t, E'u]$ for example, and that $x$'s binding has not yet been reduced to a value.

However, in presence of data constructors that permit building cyclic values not all such cases are failures. For example the term $\mathtt{let\ rec}\ x = \mathtt{Pair}\,(x, t)\ \mathtt{in}\ x$ can be decomposed into $E[x]$ to isolate the occurrence of $x$ as the first member of the pair. This occurrence of $x$ is in reducible position, but there is no $v$ such that $(x = v) \stackrel{\mathsf{ctx}}{\in} E$, unless $t$ is already a weak value.

To characterize failures during recursive evaluation, we propose to restrict ourselves to *forcing contexts*, denoted $E_{\mathsf{f}}$, that must access or return the value of their hole. A variable in a forcing context that cannot be looked up in the context is a dynamic failure: we are forcing the value of a variable that has not yet been evaluated. If a term contains such a variable in lookup position, we call it a *vicious* term.

Figure 5 gives a precise definition of these failure terms.

Mismatches are characterized by *head frames*, context fragments that would form a $\beta$-redex if they were plugged a value of the correct type. A term of the form $H[v]$ that is stuck for head-reduction is a constructor-destructor mismatch.

The definition of forcing contexts $E_\mathsf{f}$ takes into account the fact that recursive value bindings remain, floating around, in the evaluation context. A forcing frame $F_\mathsf{f}$ is a context fragment that forces evaluation of its variable; it would be tempting to say that a forcing context is necessarily of the form $\square$ or $E[F_\mathsf{f}]$, but for example $F_\mathsf{f}[\texttt{let rec } B \texttt{ in } \square]$ must also be considered a forcing context.

Note that, due to the flexibility we gave to the evaluation order, mismatches and vicious terms need not be stuck: they may have other reducible positions in their evaluation context. In fact, a vicious term failing on a variable $x$ may reduce to a non-vicious term if the binding of $x$ is reduced to a value.

### 5.3 Soundness

The proofs for these results are in Appendix B.

**Lemma 1 (Forcing modes).**
*If* $\Gamma, x : m_x \vdash E_\mathsf{f}[x] : m$ *with* $m \succeq \mathsf{Return}$*, then also* $m_x \succeq \mathsf{Return}$*.*

**Theorem 1 (Vicious).** $\emptyset \vdash t : \mathsf{Return}$ *never holds for* $t \in \mathsf{Vicious}$*.*

**Theorem 2 (Subject reduction).** *If* $\Gamma \vdash t : m$ *and* $t \to t'$ *then* $\Gamma \vdash t' : m$*.*

**Corollary 1.** $\mathsf{Return}$*-typed programs cannot go vicious.*

## 6 Extension to a full language

We now discuss the extension of our typing rules to the full OCaml language, whose additional features (such as exceptions, first-class modules and GADTs) contain some subtleties that needed special care.

### 6.1 The size discipline

The OCaml compilation scheme, one of several possible ways of treating recursive declarations, proceeds by reserving heap blocks for the recursively-defined values, and using the addresses of these heap blocks (which will eventually contain the values) as dummy values: it adds the addresses to the environment and computes the values accordingly. If no vicious term exists, the addresses are never dereferenced during evaluation, and evaluation produces “correct” values. Those correct values are then moved into the space occupied by the dummies, so that the original addresses contain the correct result.

This strategy depends on knowing how much space to allocate for each value. Not all OCaml types have a uniform size; for example, variants (sum types) may contain constructors with different arities, resulting in different in-memory size, and the size of a closure depends on the number of free variables.

After checking that mutually-recursive definitions are meaningful using the rules we described, the OCaml compiler checks that it can realize them, by trying to infer a static size for each value. It then accepts to compile each declaration if either:

– it has a static size, or
– it doesn't have a statically-known size, but its usage mode of mutually-recursive definitions is always Ignore

(The second category corresponds to detecting some values that are actually non-recursive and lifting them out. Such non-recursive values often occur in standard programming practice, when it is more consistent to declare a whole block as a single `let rec` but only some elements are recursive.)

This static-size test may depend on lower-level aspects of compilation, or at least value representation choices. For example,

```
if p then (fun x -> x) else (fun x -> not x)
```

has a static size (both branches have the same size), but

```
if p then (fun x -> x + 1) else (fun x -> x + offset)
```

does not: the second function depends on a free variable `offset`, so it will be allocated in a closure with an extra field. (While `not` is also a free variable, it is a statically-resolvable reference to a global name.)

### 6.2 Dynamic representation checks: float arrays

OCaml uses a dynamic representation check for its polymorphic arrays: when the initial array elements supplied at array-creation time are floating-point numbers, OCaml chooses a specialized, unboxed representation for the array.

Inspecting the representation of elements during array creation means that although array construction looks like a guarding context, it is often in fact a dereference. There are three cases to consider: first, where the element type is statically known to be `float`, array elements will be unboxed during creation, which involves a dereference; second, where the element type is statically known not to be `float`, the inspection is elided; third, when the element type is not statically known the elements will be dynamically inspected – again a dereference.

The following program must be rejected, for example:

```
let rec x = (let u = [|y|] in 10.)
    and y = 1.
```

since creating the array `[|y|]` will unbox the element `y`, leading to undefined behavior if `y` – part of the same recursive declaration – is not yet initialized.

### 6.3 Exceptions and first-class modules

In OCaml, exception declarations are generative: if a functor body contains an exception declaration then invoking the functor twice will declare two exceptions with incompatible representations, so that catching one of them will not interact with raising the other.

Exception generativity is implemented by allocating a memory cell at functor-evaluation time (in the representation of the resulting module); and including the address of this memory cell as an argument of the exception payload. In particular, creating an exception value `M.Exit 42` may dereference the module `M` where `Exit` is declared.

Combined with another OCaml feature, first-class modules, this generativity can lead to surprising incorrect recursive declarations, by declaring a module with an exception and using the exception in the same recursive block.

For instance, the following program is unsound and rejected by our analysis:

```
module type T = sig exception A of int end

let rec x = (let module M = (val m) in M.A 42)
and (m : (module T)) = (module (struct exception A of int end) : T)
```

In this program, the allocation of the exception value `M.A 42` dereferences the memory cell generated for this exception in the module `M`; but the module `M` is itself defined as the first-class module value `(m : (module T))`, part of the same recursive nest, so it may be undefined at this point.

(This issue was first pointed out by Stephen Dolan.)

### 6.4 GADTs

The original syntactic criterion for OCaml was implemented not directly on surface syntax, but on an intermediate representation quite late in the compiler pipeline (after typing, type-erasure, and some desugaring and simplifications). In particular, at the point where the check took place, exhaustive single-clause matches such as `match t with x -> ...` or `match t with () -> ...`) had been transformed into direct substitutions.

This design choice led to programs of the following form being accepted:

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

While this seems entirely innocuous, it becomes unsound with the addition of GADTs to the language:

```
type (_, _) eq = Refl : ('a, 'a) eq
let universal_cast (type a) (type 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 `universal_cast` violate type safety.

### 6.5 Laziness

OCaml's evaluation is eager by default, but it supports an explicit form of lazy evaluation: the programmer can write `lazy e` and `force e` to delay and force the evaluation of an expression.

The OCaml implementation performs a number of optimizations involving `lazy`. For example, when the argument of `lazy` is a trivial syntactic value (variable or constant), since eager and lazy evaluation usually behave equivalently, the compiler picks eager evaluation as an optimization to avoid thunk allocation.

However, for recursive definitions eager and lazy evaluation are not equivalent, and so the recursion check must treat `lazy trivialvalue` as if the `lazy` were not there. For example, the following recursive definition is disallowed, since the optimization described above nullifies the delaying effect of the `lazy`

```
let rec x = lazy y and y = ...
```

while the following definition is allowed by the check, since the argument to `lazy` is not sufficiently trivial to be subject to the optimization:

```
let rec x = lazy (y+0) and y = ...
```

Our typing rule for `lazy` takes this into account: "trivial" thunks are checked in mode $\mathsf{Return}$ rather than $\mathsf{Delay}$.

## 7 Conclusion

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

### 7.1 Discussion

*Right-to-left in general* Typing rules are a specialized declarative language to describe and justify various computational processes related to a type system (type checking, type inference, elaboration, etc.). Our right-to-left reading is one possible way to describe the static analysis we are capturing, which could also be described in many other ways: as pseudocode, as a fixpoint of equations, through a denotational semantics, etc. In general we believe that right-to-left readings can give a nice, compact, declarative presentation of certain demand analyses, in a language that type designers are already familiar with.

*Future work on compilation* A natural question for this work is whether the access-mode derivations we build in our safety check can inform the compilation strategy for recursive values. Our own work has concentrated on satefy, but there is ongoing work by other people on the compilation method for recursive values, which partly goes in this direction.

### 7.2 Related work

*Backward analyses* Our right-to-left reading is a particular case of *backward analysis*, as presented for example by Hughes (1987). A lot of work on backward analysis for functional programs has a denotational flavor, while we stick to a type system, giving a more declarative presentation. (Thanks to Joachim Breitner for the reference.)

*Degrees* Boudol (2001) introduces the notion of "degree" $\alpha \in \{0,1\}$ to statically analyze recursion in object-oriented programs (recursive objects, lambda-terms). Degrees refine a standard ML-style type system for programs, with a judgment of the form $\Gamma \vdash t : \tau$ where $\tau$ is a type and $\Gamma$ gives both a type and a degree for each variable. A context variable has degree 0 if it is required to evaluate the term (related to our Dereference), and 1 if it is not required (related to our Delay). Finally, function types are refined with a degree on their argument: a function of type $\tau^0 \to \tau'$ accesses its argument to return a result, while a $\tau^1 \to \tau'$ function does not use its argument right away, for example a curried function $\lambda x.\, \lambda y.\, (x, y)$ – whose argument is used under a delay in its body $\lambda y.\, (x, y)$. Boudol uses this reasoning to accept a definition such as `let rec obj = class_constructor obj params`, arising from object-oriented encodings, where `class_constructor` has a type $\tau^0 \to \dots$.

Our system of mode is finer-grained than the binary degrees of Boudol; in particular, we need to distinguish Dereference and Guard to allow cyclic data structure constructions.

On the other hand, we do not reason about the use of function arguments at all, so our system is much more coarse-grained in this respect. In fact, refining our system to accept `let rec obj = constr obj params` would be incorrect for our use-case in the OCaml compiler, whose compilation scheme forbids passing yet-uninitialized data to a function.

In a general design aiming for maximal expressiveness, access modes should refine ML types; in Boudol's system, degrees are interlinked with the type structure in function types $\tau^\alpha \to \tau'$, but one could also consider pair types of the form ${\tau_1}^{\alpha_1} \times {\tau_2}^{\alpha_2}$, etc. In our simpler system, there are no interaction between value shapes (types) and access modes, so we can forget about types completely, a nice conceptual simplification. Our formalization will be done entirely in an untyped fragment of ML.

*Compilation* Hirschowitz, Leroy, and Wells (2003, 2009) discuss the space of compilation schemes for recursive value definitions, and prove the correctness of a compilation scheme similar to one used by the OCaml compiler, using in-place update to tie the knot after recursive bindings are evaluated. Their source language has `let rec` bindings and a source-level operational semantics, based on floating bindings upwards in the term (similar to explicit substitutions or local thunk stores). Their target language can talk about uninitialized memory cells and their update, and a mutable-store operational semantics.

In the present work, we do not formalize a compilation scheme for recursive definitions, we only prove our static analysis correct with respect to a source-level operational semantics.

While they are presenting a lambda-calculus, these works were concerned with recursive modules and mixin modules in ML languages – as other related work below. Recursive modules are used when programming at large, where programmers are willing to introduce cyclic dependencies in subtle, non-local ways, which requires fine-grained checks.

We only consider term-level cyclic value definitions, a simpler problem domain where less static sophistication is demanded. In fact, we do not aim at accepting substantially more recursive definitions than the previous OCaml syntactic check, only to be more trustworthy.

*Name access as an effect* Dreyer (2004) proposes to track usage of recursively-defined variables as an effect, and designs a type-and-effect system whose effects annotations are sets of abstract names, maintained in one-to-one correspondence with `let rec`-bound variables. The construction `let rec X▷x :` $\tau$ `= e` introduces the abstract type-level name `X` corresponding to the recursive variable `x`. This recursive variable is made available in the scope of the right-hand-side `e :` $\tau$ at the type `box(X,`$\tau$`)` instead of $\tau$ (reminding us of guardedness modalities). Any dereference of `x` must explicitly "unbox" it, adding the name `X` to the ambient effect.

This system is very powerful, but we view it as a core language rather than a surface language: encoding a specific usage pattern may require changing the types of the components involved, to introduce explicit `box` modalities:

- When one defines a new function from $\tau$ to $\tau'$, one needs to think about whether it may be later used with still-undefined recursive names as argument – assuming it indeed makes delayed uses of its argument. In that case, one should use the usage-polymorphic type function type $\forall X.\mathsf{box}(X,\tau) \to \tau'$ instead of the simple function type $\tau \to \tau'$. (It is possible to inject $\tau$ into $\mathsf{box}(X,\tau)$, so this does not restrict non-recursive callers.)
- One could represent cyclic data such as `let rec ones = 1 :: ones` in this system, but it would require a non-modular change of the type of the list-cell constructor from $\forall\alpha.\alpha \to \mathsf{List}(\alpha) \to \mathsf{List}(\alpha)$ to the box-expecting type $\forall\alpha.\alpha \to \forall X.\mathsf{box}(X,\mathsf{List}(\alpha)) \to \mathsf{List}(\alpha)$ .

In particular, one cannot directly use typability in this system as a static analysis for a source language; this work needs to be complemented by a static analysis such as ours, or the safety has to be proved manually by the user placing box annotations and operations.

*Graph typing* Hirschowitz also collaborated on static analyses for recursive definitions in Hirschowitz and Lenglet (2005); Bardou (2005). The design goal was a simpler system than existing work aiming for expressiveness, with inference as simple as possible.

As a generalization of Boudol's binary degrees they use compactified numbers $\mathbb{N} \cup \{-\infty, \infty\}$. The degree of a free variable "counts" the number of subsequent $\lambda$-abstractions that have to be traversed before the variable is used; $x$ has degree 2 in $\lambda y.\, \lambda z.\, x$. A $-\infty$ is never safe, it corresponds to our $\mathsf{Dereference}$ mode. 0 conflates our $\mathsf{Guard}$ and $\mathsf{Return}$ mode (an ad-hoc syntactic restriction on right-hand-sides is used to prevent under-determined definitions), the $n+1$ are fine-grained representations of our $\mathsf{Delay}$ mode, and finally $+\infty$ is our $\mathsf{Ignore}$ mode.

Another salient aspect of their system is the use of "graphs" in the typing judgment: a use of `y` within a definition `let x = e` is represented as an edge from `y` to `x` (labeled by the usage degree), in a constraint graph accumulated in the typing judgment. The correctness criterion is formulated in terms of the transitive closure of the graph: if `x` is later used somewhere, its usage implies that `y` also needs to be initialized in this context.

Our work does not need such a transitive-computation device, as our `let` rule uses a simple form of mode substitution to propagate usage information. One contribution of our work is to show that a more standard syntactic approach can replace the graph representation.

Finally, their static analysis mentions the in-memory size of values, which needs to be known statically, in the OCaml compilation scheme, to create uninitialized memory blocks for the recursive names before evaluating the recursive definitions. Our type system does not mention size at all, it is complemented by an independent (and simpler) analysis of static-size deduction, which is outside the scope of the present formalization, but described briefly in Section 6.1 (The size discipline).

*F♯* Syme (2006) proposes a simple translation of mutually-recursive definitions into lazy/force constructions:. For example, `let rec` $x$ `=` $t$ `and` $y$ `=` $u$ is turned into

$$\begin{array}{l}\texttt{let rec } x_{\mathsf{thunk}} \texttt{ = lazy } (t[\mathsf{force}\ x_{\mathsf{thunk}}/x, \mathsf{force}\ y_{\mathsf{thunk}}/y]) \\ \quad\quad \texttt{and } y_{\mathsf{thunk}} \texttt{ = lazy } (u[\mathsf{force}\ x_{\mathsf{thunk}}/x, \mathsf{force}\ y_{\mathsf{thunk}}/y]) \\ \texttt{let } x \texttt{ = force } x_{\mathsf{thunk}} \\ \texttt{let } y \texttt{ = force } y_{\mathsf{thunk}}\end{array}$$

With this semantics, evaluation happens on-demand, which the recursive definitions evaluated at the time where they are first accessed. This implementation is very simple, but it turns vicious definitions into dynamic failures – handled by the lazy runtime which safely raises an exception. However, this elaboration cannot support cyclic data structures: The translation of `let rec ones = 1 :: ones` fails at runtime:

$$\texttt{let rec ones}_{\mathsf{thunk}} \texttt{ = lazy (1 :: force ones}_{\mathsf{thunk}}\texttt{)}$$

Nowadays, F♯ provides an ad-hoc syntactic criterion, the "Recursive Safety Analysis" (Syme, 2012), roughly similar to the previous OCaml syntactic criterion, that distinguishes "safe" and "unsafe" bindings in a mutually-recursive group; only the latter are subjected to the thunk-introducing translation

Finally, the implementation also performs a static analysis to detect some definitions that are bound to fail – it over-approximates safety by considering ignoring occurrences within function abstractions, objects or lazy thunks, even if those delaying terms may themselves be called/accessed/forced at definition time. We believe that we could recover a similar analysis by changing our typing rules for our constructions – but with the OCaml compilation scheme we must absolutely remain sound.

*Operational semantics* Hirschowitz, Leroy, and Wells (2003, 2009) give operational semantics for a source-level language (floating `let rec` bindings) and a small-step semantics for their compilation-target language with mutable stores. Boudol and Zimmer (2002) and Dreyer (2004) use an abstract machine. Syme (2006) translates recursive definitions into lazy constructions, so the usual thunk-store semantics of laziness can be used to interpret recursive definitions. Finally, Nordlander, Carlsson, and Gill (2008) give the simplest presentation of a source-level semantics we know of; we extend it with algebraic datatypes and pattern-matching, and use it as a reference to prove the soundness of our analysis.

One inessential detail in which the semantics often differ is the evaluation order of mutually-recursive right-hand-sides. Many presentations enforce an arbitrary (e.g. left-to-right) evaluation order. Some systems (Syme, 2006; Nordlander, Carlsson, and Gill, 2008) allow a reduction to block on a variable whose definition is not yet evaluated, and go evaluate it in turn; this provides the "best possible order" for the user. Another interesting variant would be to say that the reduction order is unspecified, and that trying to evaluate an uninitialized is always a fatal error / stuck term; this provides the "worst possible order", failing as much as possible; as far as we know, the previous work did not propose it, although it is a simple presentation change. Most static analyses are evaluation-order-independent, so they are sound and complete with respect to the "worst order" interpretation.

# Bibliography


Beniamino Accattoli and Delia Kesner. The structural lambda-calculus. working paper or preprint, October 2010.

Romain Bardou. Typage des modules récursifs en caml. Technical report, ENS Lyon, July 2005.

Gérard Boudol. The recursive record semantics of objects revisited. In *Programming Languages and Systems*, 2001.

Gérard Boudol and Pascal Zimmer. Recursion in the call-by-value lambda-calculus. In *FICS*, 2002.

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

Derek Dreyer. A type system for well-founded recursion. In *POPL*, 2004.

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

Tom Hirschowitz and Sergueï Lenglet. A practical type system for generalized recursion. Technical report, ENS Lyon, 2005.

Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *PPDP*, 2003.

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.

John Hughes. Backwards analysis of functional programs. In *Partial Evaluation and Mixed Computation*, 1987.

Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150, 2017.

Oleg Kiselyov. The design and implementation of BER MetaOCaml - system description. In *FLOPS*, 2014.

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

Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *ML Workshop*, March 2006.

Don Syme. The fsharp language reference, versions 2.0 to 4.1, section 14.6.6, recursive safety analysis, 2012.

## A Properties

The following technical results can be established by simple inductions on typing derivations, without any reference to an operational semantics.

**Lemma 2 (Ignore inversion).** *$\Gamma \vdash t : \mathsf{Ignore}$ is provable with only* Ignore *in $\Gamma$.*

**Lemma 3 (Delay inversion).** *$\Gamma \vdash t : \mathsf{Delay}$ holds exactly when $\Gamma$ maps all free variables of $t$ to* Delay *or* Ignore*.*

**Lemma 4 (Dereference inversion).** *$\Gamma \vdash t : \mathsf{Dereference}$ holds exactly when $\Gamma$ maps all free variables of $t$ to* Dereference*.*

**Lemma 5 (Environment flow).** *If a derivation $\Gamma \vdash t : m$ contains a sub-derivation $\Gamma' \vdash t' : m'$, then $\forall x \in \Gamma,\ \Gamma(x) \succeq \Gamma'(x)$.*

**Lemma 6 (Weakening).** *If $\Gamma \vdash t : m$ holds then $\Gamma + \Gamma' \vdash t : m$ also holds.*

(Weakening would not be admissible if our variable rule imposed Ignore on the rest of the context.)

**Lemma 7 (Substitution).** *If $\Gamma, x : m_u \vdash t : m$ and $\Gamma' \vdash u : m_u$ hold, then $\Gamma + \Gamma' \vdash t[u/x] : m$ holds.*

**Lemma 8 (Subsumption elimination).** *Any derivation in the system can be rewritten so that the subsumption rule is only applied with the variable rule as premise.*

**Theorem 3 (Principal environments).** *Whenever both $\Gamma_1 \vdash t : m$ and $\Gamma_2 \vdash t : m$ hold, then $\min(\Gamma_1, \Gamma_2) \vdash t : m$ also holds.*

*Proof.* The proof first performs subsumption elimination on both derivations, and then by simultaneous induction on the results. The elimination phase makes proof syntax-directed, which guarantees that (on non-variables) the same rule is always used on both sides in each derivation.

This results tells us that whenever $\Gamma \vdash t : m$ holds, then it holds for a minimal environment $\Gamma$ – the minimum of all satisfying $\Gamma$.

**Definition 1 (Minimal environment).** *$\Gamma$ is minimal for $t : m$ if $\Gamma \vdash t : m$ and, for any $\Gamma' \vdash t : m$ we have $\Gamma \preceq \Gamma'$.*

In fact, we can give a precise characterization of "minimal" derivations, that uniquely determines the output of our right-to-left algorithm.

**Definition 2 (Minimal binding rule).** *An application of the binding rule is* minimal *exactly when the choice of $\Gamma'_i$ is the least solution to the recursive equation in its third premise.*

**Definition 3 (Minimal derivation).** *A derivation is* minimal *if it does not use the subsumption rule, each binding rule is minimal and, in the conclusion $\Gamma \vdash x : m$ of each variable rule, $\Gamma$ is minimal for $x : m$.*

**Definition 4 (Minimization).** *Given a derivation $D :: \Gamma \vdash t : m$, we define the (minimal) derivation* $\mathsf{minimal}(D)$ *by:*

- *Turning each binding rule into a minimal version of this binding rule – this may require applying Lemma 6 (Properties) to the* `let rec` *derivation below.*
- *Performing subsumption-elimination to get another derivation of $\Gamma \vdash t : m$.*
- *Replacing the context of each variable rule by the minimal context for this variable, which gives a minimal derivation of $\Gamma_m \vdash t : m$ with $\Gamma_m \preceq \Gamma$ (this does not introduce new subsumptions).*

**Lemma 9 (Stability).** *If $D$ is a minimal derivation, then* $\mathsf{minimal}(D) = D$.

**Lemma 10 (Determinism).** *If $D_1 :: \Gamma_1 \vdash t : m$ and $D_2 :: \Gamma_2 \vdash t : m$, then* $\mathsf{minimal}(D_1)$ *and* $\mathsf{minimal}(D_2)$ *are the same derivation.*

**Corollary 2 (Minimality).** *The environment $\Gamma$ of a derivation $\Gamma \vdash t : m$ is minimal for $t : m$ if and only if $\Gamma \vdash t : m$ admits a minimal derivation.*

*Proof.* If $\Gamma$ is minimal for $t : m$, then the context $\Gamma_m \preceq \Gamma$ obtained by minimization must itself be $\Gamma$.

Conversely, if a derivation $D_m :: \Gamma \vdash t : m$ is minimal, then all other derivations $\Gamma' \vdash t : m$ have $D_m$ as minimal derivation by Lemma 9 (Properties) and Lemma 10 (Properties), so $\Gamma \preceq \Gamma'$ holds.

**Theorem 4 (Localization).** *$\Gamma \vdash t : m'$ implies $m\,[\Gamma] \vdash t : m\,[m']$.*

*Furthermore, if $\Gamma$ is minimal for $t : m'$, then $m\,[\Gamma]$ is minimal for $t : m\,[m']$.*

*Proof.* The proof proceeds by direct induction on the derivation, and does not change its structure: each rule application in the source derivation becomes the same source derivation in the result. In particular, minimality of derivations is preserved, and thus, by Corollary 2 (Properties), minimality of environments is preserved.

Besides associativity of mode composition, many cases rely on the fact that external mode composition preserves the mode order structure: $m'_1 \prec m'_2$ implies $m\,[m'_1] \prec m\,[m'_2]$, and $\max(m\,[m'_1]\,, m\,[m'_2])$ is $m\,[\max(m'_1, m'_2)]$.

## B Proofs for Section 5.3 (Soundness)

**Lemma 11 (1: Soundness).**
*If $\Gamma, x : m_x \vdash E_{\mathsf{f}}[x] : m$ with $m \succeq \mathsf{Return}$, then also $m_x \succeq \mathsf{Return}$.*

*Proof.* $E_{\mathsf{f}}$ may be of the form $L$ or $E[F_{\mathsf{f}}[L]]$

In the case of binding contexts $L$ we have $m_x = \mathsf{Return}$ by construction.

In the case with a forcing frame, $E_{\mathsf{f}} = E[F_{\mathsf{f}}[L]]$, let us call $m_E$ the mode of the hole of $E$. It is immediate that the mode imposed by $L$ on its hole is $\mathsf{Return}$, and that the mode imposed by $F_{\mathsf{f}}$ on its own hole is $\mathsf{Dereference}$, so the total mode $m_x$ is $m_E\,[\mathsf{Dereference}\,[\mathsf{Return}]]$. We can prove by an easy induction on $E$ that $m_E$ is not $\mathsf{Ignore}$ or $\mathsf{Delay}$ – those are not evaluation contexts, so we have $m_E \succeq \mathsf{Guard}$. We conclude by monotonicity of mode composition:

$$m_x = m_E\,[\mathsf{Dereference}\,[\mathsf{Return}]] \succeq \mathsf{Guard}\,[\mathsf{Dereference}\,[\mathsf{Return}]] = \mathsf{Dereference}$$

**Theorem 5 (1: Soundness).** $\emptyset \vdash t : \mathsf{Return}$ *never holds for* $t \in \mathsf{Vicious}$.

*Proof.* Given $\vdash t : \mathsf{Return}$, let us assume that $t$ is $E[x]$ with no value binding for $x$ in $E$, and show that $E$ is not a forcing context.

We implicitly assume that all terms are well-scoped, so the absence of value binding means that $x$ occurs in a `let rec` binding still being evaluated somewhere in $E$: $E[x]$ is of the form

$$E[x] = E_{\mathsf{out}}[t_{\mathsf{rec}}] \qquad\qquad t_{\mathsf{rec}} = (\texttt{let rec}\ b, y = E_{\mathsf{in}}[x], b'\ \texttt{in}\ u)$$

where $x$ is bound in $b$, $b'$ or is $y$ itself.

Given our `let rec` typing rule (see Figure 3), the typing derivation for $t$ contains a sub-derivation for $t_{\mathsf{rec}}$ of the form

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

In particular, the premise for $E_{\mathsf{in}}[x]$ is of the form $\Gamma, (x_j : m_j)^j \vdash E_{\mathsf{in}}[x] : \mathsf{Return}$ with $(x_j \preceq \mathsf{Guard})^j$, and in particular $x \preceq \mathsf{Guard}$ so $x \not\succeq \mathsf{Return}$.

By Lemma 1 (Soundness), $E_{\mathsf{in}}$ cannot be a forcing context, and in consequence $E$ is not forcing either.

**Theorem 6 (2: Soundness).** *If* $\Gamma \vdash t : m$ *and* $t \to t'$ *then* $\Gamma \vdash t' : m$.

*Proof.* We reason by inversion on the typing derivation of redexes, first for head-reduction $t \to^{\mathsf{hd}} t'$ and then for reduction $t \to t'$.

*Head reduction* We only show the head-reduction case for functions; pattern-matching is very similar. We have:

$$\frac{\dfrac{\dfrac{\Gamma_t, x : m_x \vdash t : m\,[\mathsf{Dereference}]\,[\mathsf{Delay}]}{\Gamma_t \vdash \lambda x.\, t : m\,[\mathsf{Dereference}]}}{\Gamma_t \vdash L[\lambda x.\, t] : m\,[\mathsf{Dereference}]} \qquad \Gamma_v \vdash v : m\,[\mathsf{Dereference}]}{\Gamma_t + \Gamma_v \vdash L[\lambda x.\, t]\ v : m}$$

By associativity, $m\,[\mathsf{Dereference}]\,[\mathsf{Delay}]$ is the same as $m\,[\mathsf{Dereference}]$.

By subsumption, $\Gamma_t, x : m_x \vdash t : m\,[\mathsf{Dereference}]$ implies $\Gamma_t, x : m_x \vdash t : m$.

To conclude by using Lemma 7 (Properties), we must reconcile the mode of the argument $v : m\,[\mathsf{Dereference}]$ with the (apparently arbitrary) mode $x : m_x$ of the variable. We reason by an inelegant case distinction.

- If $m\,[\mathsf{Dereference}]$ is $\mathsf{Dereference}$, then by inversion (Lemma 4) either $m_x$ is $\mathsf{Dereference}$ (problem solved) or $x$ does not occur in $t$ (no need for the substitution lemma).
- If $m\,[\mathsf{Dereference}]$ is not $\mathsf{Dereference}$, then $m$ must be $\mathsf{Ignore}$ or $\mathsf{Delay}$. If it is $\mathsf{Ignore}$, inversion (Lemma 2) directly proves our goal. If it is $\mathsf{Delay}$, then by inversion (Lemma 3) $m_x$ itself can be weakened (subsuming the derivation of $t$) to be below $\mathsf{Delay}$.

*Reduction under context* Reducing a head-redex under context preserves typability by the argument above. Let us consider the lookup case.

$$\frac{(x = v) \overset{\mathsf{ctx}}{\in} E}{E[x] \to E[v]}$$

By inspecting the $(x = v) \overset{\mathsf{ctx}}{\in} E$ derivation, we find a value binding $B$ within $E$ with $x = v$, and a derivation of the form

$$\frac{(x_i : \Gamma'_i)^i \vdash \mathtt{rec}\ B \qquad (m'_i)^i \overset{\mathsf{def}}{=} (\max(m_i, \mathsf{Guard}))^i \qquad \Gamma_u, (x_i : m_i)^i \vdash u : m}{\sum (m'_i\,[\Gamma'_i])^i + \Gamma_u \vdash \mathtt{let\ rec}\ B\ \mathtt{in}\ u : m}$$

$$\frac{\begin{array}{c}(\Gamma_i, (x_j : m_{i,j})^j \vdash v_i : \mathsf{Return})^i \qquad (m_{i,j} \preceq \mathsf{Guard})^{i,j} \\ (\Gamma'_i = \Gamma_i + \sum (m_{i,j}\,[\Gamma'_j])^j)^i\end{array}}{(x_i : \Gamma_i)^i \vdash \mathtt{rec}\ (x_i = v_i)^i}$$

By abuse of notation, we will write $m_x$, $\Gamma_x$ and $\Gamma'_x$ to express the $m_i$, $\Gamma_i$ and $\Gamma'_i$ for the $i$ such that $x_i = x$.

The occurrence of $x$ in the hole of $E[\square]$ is typed (eventually by a variable rule) at some mode $m_\square$. The declaration-side mode $m_x$ was built by collecting the usage modes of all occurrences of $x$ in the `let rec` body $u$, which in particular contains the hole of $E$, so we have $m_\square \preceq m_x$ by Lemma 5 (Properties).

The binding derivation gives us a proof $\Gamma_x, \Gamma_{\mathtt{rec}} \vdash v : \mathsf{Return}$ that the binding $x = v$ was correct at its definition site, where $\Gamma_{\mathtt{rec}}$ has exactly the mutually-recursive variables $(x_i : m_i)^i$. Notice that this subderivation is completely independent of the ambient expected mode $m$.

By Theorem 4 (Properties), we can compose this within $m_\square$ to get a derivation $m_\square\,[\Gamma_x, \Gamma_{\mathtt{rec}}] \vdash v : m_\square$, that we wish to substitute into the hole of $E$. First we weaken it (Lemma 6) into the judgment $m_x\,[\Gamma_x, \Gamma_{\mathtt{rec}}] \vdash v : m_\square$.

Plugging this derivation in the hole of $E$ requires weakening the derivation of $u$ (the part of $E[\square]$ that is after the declaration of $x$) to add the environment $m_x\,[\Gamma_x, \Gamma_{\mathtt{rec}}]$. Weakening is always possible (Lemma 6), but it may change the environment of the derivation, while we need to preserve the environment of $E[x]$. Consider the following valid derivation:

$$\frac{\begin{array}{c}(x_i : \Gamma'_i)^i \vdash \mathtt{rec}\ B \qquad (m''_i)^i \overset{\mathsf{def}}{=} (\max(\max(m_i, m_x\,[\Gamma_{\mathtt{rec}}]\,(x_i)), \mathsf{Guard}))^i \\ \Gamma_u + m_x\,[\Gamma_x]\,, (x_i : m_i)^i + m_x\,[\Gamma_{\mathtt{rec}}] \vdash u[v/x] : m\end{array}}{\sum (m''_i\,[\Gamma'_i])^i + \Gamma_u + m_x\,[\Gamma_x] \vdash \mathtt{let\ rec}\ B\ \mathtt{in}\ u[v/x] : m}$$

To show that we preserve the environment of $E[x]$, we show that this derivation is not in a bigger environment than the environment of our source term:

$$\sum (m_i'' [\Gamma_i'])^i + \Gamma_u + m_x [\Gamma_x] \qquad \preceq \qquad \sum (m_i' [\Gamma_i'])^i + \Gamma_u$$

By construction we have $m_x \preceq m_x' \preceq m_x''$ and $\Gamma_x \preceq \Gamma_x'$, so $m_x [\Gamma_x] \preceq m_x'' [\Gamma_x']$ which implies

$$\sum (m_i'' [\Gamma_i'])^i + \Gamma_u + m_x [\Gamma_x] \qquad \preceq \qquad \sum (m_i'' [\Gamma_i'])^i + \Gamma_u$$

Then, notice that $\Gamma_{\mathtt{rec}}(x_i)$ is exactly $m_{x,i}$, so $m_i''$ is $\max(m_i', m_x [m_{x,i}])$. We can thus rewrite $m_i'' [\Gamma_i']$ into $m_i' [\Gamma]_i' + m_x [m_{x_i}] [\Gamma_i']$, which gives

$$\sum (m_i'' [\Gamma_i'])^i + \Gamma_u \qquad = \qquad \sum (m_i' [\Gamma_i'])^i + m_x \left[\sum (m_{x,i} [\Gamma_i'])^i\right] + \Gamma_u$$

The extra term $\sum (m_{x,i} [\Gamma_i'])^i$ is precisely the term that appears in the definition of $\Gamma_x'$ from the $(\Gamma_i)^i$, taking into account transitive mutual dependencies – indeed, when we replace $x$ by its value $v$, we replace transitive dependencies on its mutual variables by direct dependencies on occurrences in $v$. We thus have

$$\sum (m_{x,i} [\Gamma_i'])^i \preceq \Gamma_x'$$

and can conclude with

$$\begin{aligned} & \sum (m_i' [\Gamma_i'])^i + m_x \left[\sum (m_{x,i} [\Gamma_i'])^i\right] + \Gamma_u \\ \preceq & \sum (m_i' [\Gamma_i'])^i + m_x [\Gamma_x'] + \Gamma_u \\ \preceq & \sum (m_i' [\Gamma_i'])^i + \Gamma_u \end{aligned}$$