A Practical Mode System for Recursive Definitions

ALBAN REYNAUD, ENS Lyon, France
GABRIEL SCHERER, INRIA, France
JEREMY YALLOP, University of Cambridge, United Kingdom

In call-by-value languages, some mutually-recursive 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 directed into an algorithmic backwards analysis 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 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.


Additional Key Words and Phrases: recursion, call-by-value, types, semantics, ML, functional programming

ACM Reference Format:

1 INTRODUCTION
Recursion pervades functional programs. Functional programmers often start out by writing simple recursive definitions such as the Fibonacci function, shown here in OCaml:

let rec fib = fun x -> if x <= 1 then x else fib (x-1) + fib (x-2)

This definition is elegant but, alas, impractical: computing \( \text{fib} \ n \) takes time exponential in \( n \). One way to improve performance is to memoize: in place of a function, we might (recursively) define a lazy list, \( \text{lfibs} \), whose \( n \)th element represents \( \text{fib} \ n \):

let rec lfibs = lazy (\( \_ :: \) lazy (\( \_ :: \text{map2 (+)} \) lfibs (tail lfibs))))

As these definitions show, recursion is useful for defining both functions and values. However, lazy-list memoization is rarely used in eager languages, since elegance suffers from the need to make all laziness explicit. Here is a more idiomatic memoized function \( \text{mfib} \), mutually-defined with a record \( \text{mfibs} \), which pairs the function with a memo-table of its previously-computed values:
let rec mfib = fun x -> if x <= 1 then x
   else remember mfibs (x-1) + remember mfibs (x-2)
and mfibs = { f = mfib; values = empty_table () }

(The remember function retrieves previously-computed values and computes and stores new entries.)

This definition exposes both mfib and its table mfibs, risking inadvertent modification of the table.

A cautious programmer might avoid this danger by making mfibs local to mfib:

let rec mfib' = let mfibs' = { f = mfib'; values = empty_table () } in
   fun x -> if x <= 1 then x
       else remember mfibs' (x-1) + remember mfibs' (x-2)

1.1 Recursion: Expressiveness vs Safety

As this tapestry of fibs suggests, the usefulness of recursion is not limited to simple function definitions: our examples build record values, call functions and bind local names. However, in eager languages not all recursion is safe. For example, here is an unsafe eager variant of lfibs:

let rec efibs = 0 :: 1 :: map2 (+) efibs (tail efibs) (* unsafe! *)

Evaluation of efibs fails, since map2 and tail access efibs before it is fully constructed.

Existing functional languages incorporate various approaches to balancing usefulness and safety. In some languages, such as Scheme [Sperber et al. 2009], mutually-recursive bindings are evaluated immediately, and it is a run-time error for evaluation to encounter any identifier being bound. Evaluating mfibs, which refers to mfib, would produce such an error.

Other languages, such as F#, provide a kind of lazy evaluation for recursive value bindings [Syme 2006] (discussed in §8) which supports the construction of recursive objects. F#’s approach allows more programs to execute without error — it is sufficient to support mfib, though not mfib’ — but it does not entirely eliminate run-time errors from ill-formed recursive bindings. Consequently, an additional syntactic check [Syme 2005] rejects some cases of self-reference that would result in run-time errors. (However, even this additional check is not sufficient to eliminate all such errors.)

Finally, some languages, such as Standard ML [Milner et al. 1997], incorporate a more severe approach, permitting recursion only through syntactic function definitions such as fib, and statically rejecting mfib, mfib’ and efib. (Some implementations also support laziness, and allow lfibs.)

Of these design choices, Standard ML’s most fully embodies Milner’s dictum: well-typed programs do not go wrong. However, as we show in this paper, Standard ML’s treatment of recursive definitions is unnecessarily restrictive: it is possible to define a much more permissive criterion for recursive definitions that still ensures the absence of run-time errors. Our criterion allows useful definitions such as lfibs, mfib and mfib’, while rejecting incorrect programs such as efibs. We present the criterion as a mode system (proved sound with respect to an operational semantics), suitable for incorporating into a compiler — indeed, our implementation has already been merged into the mainline OCaml compiler.

1.2 OCaml Needed Fixing

Before the work described in this paper, OCaml took an approach similar to F#’s (although somewhat more precise, and based on OCaml’s existing eager semantics rather than a translation into thunks), checking for vicious recursive definitions via syntactic analysis of an intermediate representation of programs. We believe OCaml’s check as originally defined was correct, but 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 (§2.3). In conjunction

https://ocaml.org/releases/4.05/htmlman/extn.html#sec217
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 when definitions that accessed recursively-defined objects before they were initialized were allowed through undetected.

1.3 Generalized Recursion in Practice
In order to determine whether generalized recursive definitions were used in practice, we searched a large subset of packages in the OCaml software repository (OPAM) for instances of generalized recursion (i.e. recursive definitions that Standard ML would reject). At the time of our analysis in July 2020, we found 309 distinct examples of such definitions in around 74 packages. The definitions variously made use of local bindings and other local constructs (such as local exception definitions), evaluation of sub-terms, record and variant construction and laziness, in both singly- and mutually-recursive bindings. Generalized recursion definitions are used in useful data structures and important libraries that many other packages in turn depend on. We found that 1613 packages out of 2819 in the repository at the time (57%) depend on at least one of those packages using generalized recursion.

1.4 Our Analysis
The present document formally describes our analysis using a core ML language (§3). We present inference rules (§4) and study the meta-theory of the analysis. We propose a source-level operational semantics, refreshing semantics proposed in earlier works [Ariola and Felleisen 1997; Hirschowitz, Leroy, and Wells 2009; Nordlander, Carlsson, and Gill 2008] with explicit substitutions using reduction at a distance (§5), and show that our analysis is sound for this semantics. We also propose a semantics that uses mutable updates to a global store, closer to production-compiler compilation strategies (§6), for which our analysis is also sound. Finally, we discuss the challenges caused by scaling the analysis to OCaml (§7), a full-fledged functional language, in particular the delicate interactions with non-uniform value representations (§7.2), with exceptions and first-class modules (§7.3), and with Generalized Algebraic Datatypes (GADTs) (§7.4).

1.5 Adoption in OCaml
The initial version of our new system was originally released in OCaml 4.06.0 (3 Nov 2017). We formalised the new system after its release, and reworked the implementation to better match the formalisation. The updated implementation was released in OCaml 4.08.0 (13 June 2019).

Before releasing our new system, we sought to determine whether it was significantly more restrictive in practice than the previous check by running it over the packages in OPAM. This investigation did not uncover any code that was accepted by the previous check but rejected by our new system. Six major OCaml releases later there has been only one report of such a program (#7767); it was straightforward to update our system to accept it.

Of course, since we looked only at existing OCaml code, our investigation could not uncover definitions that were rejected by the previous check but accepted by our new system, which is more permissive in some cases (§2.3). However, our goal was not to increase the expressivity of OCaml’s value definitions, but to design a static check that was backwards-compatible with the previous check, while being easier to reason about and evolve in tandem with the language (§7). Our aim was to make the check as simple as possible within the constraints, not as expressive as possible.

Furthermore, moving the check from the compiler middle end into the type checker has another benefit: it 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).

1.6 Contributions

We claim the following contributions:

- We propose a new system of inference rules that captures the safety conditions for recursive definitions in an eager language (§4), previously enforced in OCaml by ad-hoc syntactic restrictions.
- We prove the analysis sound with respect to a source-level operational semantics: accepted recursive terms evaluate without vicious-circle failures (§5). Our source-level semantics is justified by a simulation result with lower-level backpatching semantics with a global store (§6), for which our analysis is also sound.
- We have implemented a checker derived from the rules, scaled up to the full OCaml language (§7) and integrated in the OCaml implementation.
- Our analysis is less fine-grained on functions than existing works (§8), thanks to a less demanding problem domain (ML functions rather than ML functors), but in exchange it provides finer-grained handling of cyclic data and an effective inference algorithm.

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

\[
\text{let rec } f = \text{fun } x \to \ldots f \ldots
\]

the recursive reference to \( f \) in the right-hand-side does not need to be evaluated to define the function value \( \text{fun } x \to \ldots \) 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 \text{Delay}.

In contrast, in the vicious definition \( \text{let rec } x = 1 + x \) evaluation of the right-hand side involves accessing the value of \( x \); we call this usage mode a \text{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 \text{Delay} and \text{Dereference}. For example, in the cyclic datatype construction \( \text{let rec } \text{obj} = \{ \text{self} = \text{obj} \} \) the recursively-bound variable \( \text{obj} \) appears on the right-hand side without being placed inside a function abstraction. However, since it appears in a “guarded” position, within the record constructor \( \{ \text{self} = - \} \), evaluation only needs to access its address, not its value. We say that the mode of use of the variable \( \text{ones} \) is \text{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 \( \text{let } y = x \text{ in } y \), for example. In such cases we say that the value is “returned” directly and use the mode \text{Return}. As with \text{Dereference}, recursive definitions that access variables at the mode \text{Return}, such as \( \text{let rec } x = x \), would be under-determined and are rejected.

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

2.2 An Inference System (and Corresponding Backwards Analysis)

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

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

This judgment uses access modes for two purposes: to classify variables, and to classify 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 the plugged context $C[t] : \text{Return}$ will contain a sub-derivation of the form $t : m$ for the term $t$.

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 backwards interpretation. Given a term $t$ and a 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:

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

The backwards 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 check the function body $t$ under the weaker mode $m{[\text{Delay}]}$, 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 $y : \text{Delay}$ as a result.

Given a family of mutually-recursive definitions $\textbf{let rec } (x_i = t_i)^{i \in I}$, 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 $(\Gamma_i \vdash t_i : \text{Return})^{i \in I}$ 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.

#### #7231: unsoundness with nested recursive bindings.

The previous check accepted the following unsafe program.

```
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.
#7215: unsoundness with GADTs. The previous check accepted the following unsafe program.

```ocaml
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.

#6939: unsoundness with float arrays.

```ocaml
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 fail with a segmentation fault when run, as explained in Section 7.2 (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 Dereference context for its elements.

#4989: inconveniently rejected program.

```ocaml
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 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 issue (#4989) dates back to 2010: 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 \((\ldots)_{i \in I}^{l_i}\) 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 often omit the index set, writing \((\ldots)^i\). Families may range over two indices (the domain is the cartesian product), for example \((t_{i,j})_{i,j}^{i,j}\).

Our syntax, judgments, and inference rules will often use families: for example, \(\text{let rec } (x_i = t_i)^i\) is a mutually-recursive definition of families \((t_i)^i\) of terms bound to corresponding variables \((x_i)^i\) — assumed distinct, following the Barendregt convention. Sometimes a family is used where a term is expected, and the interpretation should be clear: when we say "\((I_i \uplus t_i : m_i)^i\) 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.
Terms $\ni t, u ::= x, y, z$

$| \text{let rec } b \text{ in } u$

$| \lambda x. t \mid t u$

$| K(t_i)^i \mid \text{match } t \text{ with } h$

Bindings $\ni b ::= (x_i = t_i)^i$

Handlers $\ni h ::= (p_i \rightarrow t_i)^i$

Patterns $\ni p, q ::= K(x_i)^i$

Fig. 1. Core language syntax

- Datatype constructors $K(t_1, t_2, \ldots)$ 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 ($\text{match } t \text{ with } (K_i (x_{i,j})^j \rightarrow u_i)^i$), 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.

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. However, we do assume that our terms are well-scoped — note that, in $\text{let rec } (x_i = v_i)^i \text{ in } u$, the $(x_i)^i$ are in scope of $u$ but also of all the $v_i$.

Remark 1. Recursive values are a controversial feature as they break the assumption that structurally-decreasing recursive functions will terminate on all inputs. The uses we found in the wild in OCaml programs typically combine “negative” constructs (functions, lazy, records) rather than infinite lists or trees. A possible design would be to distinguish an “inductive” sub-space of recursive types whose recursive occurrences are forbidden in negative positions, and whose constructors are not given the Guard mode in our system. In another direction, Jeannin, Kozen, and Silva [2017] propose language extensions to make it easier to operate over cyclic structures.

4 A MODE SYSTEM FOR RECURSIVE DEFINITIONS

4.1 Access/Usage Modes

Figure 2 defines the access/usage 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 $\text{let rec } x = K(x)$.

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 $\text{let rec } x = x$ and $\text{let rec } x = \text{let } y = x \text{ in } y$, the rightmost occurrence of $x$ is in Return context.

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 8. Our modes are a degenerate case, a refinement of uni-typed ML.
Modes \( \ni m := \) Ignore
| Delay
| Guard
| Return
| Dereference

Mode order:
Ignore \(<\) Delay \(<\) Guard \(<\) Return \(<\) Dereference

Mode composition rules:
- \( \text{Ignore} [m] = \text{Ignore} = m[\text{Ignore}] \)
- \( \text{Delay} [m > \text{Ignore}] = \text{Delay} \)
- \( \text{Guard} [\text{Return}] = \text{Guard} \)
- \( \text{Guard} [m \neq \text{Return}] = m \)
- \( \text{Return} [m] = m \)
- \( \text{Dereference} [m > \text{Ignore}] = \text{Dereference} \)

Mode composition as a table:

\[
\begin{array}{cccccc}
  m & m' & \text{Ignore} & \text{Delay} & \text{Guard} & \text{Return} & \text{Dereference} & m \\
\hline
  \text{Ignore} & \text{Ignore} & \text{Ignore} & \text{Ignore} & \text{Ignore} & \text{Ignore} & \text{Ignore} \\
  \text{Delay} & \text{Ignore} & \text{Delay} & \text{Delay} & \text{Delay} & \text{Dereference} \\
  \text{Guard} & \text{Ignore} & \text{Delay} & \text{Guard} & \text{Guard} & \text{Dereference} \\
  \text{Return} & \text{Ignore} & \text{Delay} & \text{Guard} & \text{Return} & \text{Dereference} \\
  \text{Dereference} & \text{Ignore} & \text{Delay} & \text{Dereference} & \text{Dereference} & \text{Dereference} \\
\end{array}
\]

Fig. 2. Access/usage modes and operations

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. \text{match} \emptyset with \( h \) is a Dereference context.

Remark 2 (Discarding). The Guard mode is also used for subterms whose result is discarded by the evaluation of their context. For example, the hole \( \emptyset \) is in a Guard context in \((\text{let } x = \emptyset \text{ in } u)\), 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 \( m < m' \) 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, \( t \emptyset \) is a Dereference context, since the function \( t \) may access its argument in arbitrary ways, while \( \lambda x. \emptyset \) is a Delay context.

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

Continuing the example above, the context \( t (\lambda x. \emptyset) \), formed by composing \( t \emptyset \) and \( \lambda x. \emptyset \), 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 contrast, the context \( \lambda x. (t \emptyset) \), formed by composing \( \lambda x. \emptyset \)
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 write overlapping environments, taking the maximum mode for each variable. We sometimes use family notation for environments, writing \( \Gamma \) 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 \( \Gamma_i \) to indicate the disjoint union of the members, and \( \sum (\Gamma_i) \) for the non-disjoint merge of a family of environments.

### 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 \( (\Gamma_i)^j \) to indicate the disjoint union of the members, and \( \sum (\Gamma_i) \) 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.

Section 4.3 (Examples) contains examples of mode judgments in our system, corresponding to recursive definitions that are accepted or rejected. Looking at those examples in parallel may help understand some of the inference rules, in particular for \texttt{let rec}.

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 declarative presentation, we let the rest of the environment \( \Gamma \) be arbitrary; we could also have imposed that it map all variables to \texttt{Ignore}. Our algorithmic check returns the “least demanding” environment \( \Gamma \) for all satisfiable judgments, so it uses \texttt{Ignore} in any case.

We have a subsumption rule; for example, if we want to check \( t \) under the mode \texttt{Guard}, it is always sound to attempt to check it under the stronger mode \texttt{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 \( m > m' \) holds, while we might expect \( m \leq m' \). This comes from the fact that our backwards reading is opposite to the usual reading direction of type judgments, and influenced our order definition. When \( m > m' \) 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 \( m \) [\texttt{Delay}]. The necessary constraints \( \Gamma \) are returned, after removing the constraint over \( x^3 \).

The application rule checks both the function and its argument in a 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 Guard context, rather than Dereference.

Pattern-matching. The rule for \texttt{match} \( t \) with \( h \) relies on a different clause judgment \( \Gamma \vdash_{cl} h : m \) that checks each clause in turn and merges their environments. On a single clause \( K (x_i)^I \rightarrow u \), 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.\(^4\)

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

Second, the \texttt{let rec} \( b \) in \( u \) rule of the term judgment takes these \( \Gamma_i \) and uses them under a composition \( m_i^I [\Gamma_i] \), 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 \( m_i^I \) is a combination of the usage mode in the body \( u \) and \texttt{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 2 (Discarding).

Deriving a simple \texttt{let} rule. Before we delve into the more general rule for mutually-recursive definitions, let us mention the particular case of a single, non-recursive definition \texttt{let} \( x = t \) in \( u \).

\(^3\)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 8), we could use the mode \( m_x \) in a richer return mode \( m_x \rightarrow m \).

\(^4\)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 enrich the datatype of the pattern scrutinee.
A Practical Mode System for Recursive Definitions

The general rule simplifies itself into the following:

\[ \Gamma_i \vdash t : \text{Return} \quad m' \overset{\text{def}}{=} \max(m_x, \text{Guard}) \quad \Gamma_u, x : m_x \vdash u : m \]

\[ m'[\Gamma_i] + \Gamma_u \vdash \lambda x = t \text{ in } u : m \]

**Binding judgment and mutual recursion.** The binding judgment \((x_i : \Gamma_i)^{\text{def}} \vdash \text{rec } b\) is independent of the ambient context and usage 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 \((x_j : m_{i,j})^{\text{def}}\) 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 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 (An Inference System (and Corresponding Backwards Analysis)).

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 \(x_i = e_j\) uses the mutually-defined variable \(x_j\) under the mode \(m_{i,j}\), then we ask that the final environment \(\Gamma_i'\) for \(e_j\) contains what you need to use \(e_j\) under the mode \(m_{i,j}\), that is \(m_{i,j}[\Gamma_j]\). 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 Guard, we can show that \(m_{i,j}[\Gamma_j] \leq \Gamma_j\). In particular, if we have a single recursive binding, we have \(\Gamma_i \geq m_{i,i}[\Gamma_i]\), so the third premise is equivalent to just \(\Gamma_i' \overset{\text{def}}{=} \Gamma_i\); the \(\Gamma_i'\) and \(\Gamma_i\) only differ for non-trivial mutual recursion.

**Unique minimal environment.** In Appendix A (Properties of our Typing Judgment) in the extended version we develop some direct meta-theoretic properties of our inference rules. We summarize here the key results. For each \(t : m\), there exists a minimal environment \(\Gamma\) such that \(\Gamma \vdash t : m\) holds.

**Theorem 1 (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.

We also define minimal derivations, which restrict the non-determinism in the variable and binding rules, and the way subsumption may be used. Minimal derivations have minimal environments and a syntax-directed structure. They precisely characterize the behavior of our algorithm, implemented in the OCaml compiler: given \(t : m\), it constructs a minimal derivation of the form \(\Gamma \vdash t : m\), and returns the environment \(\Gamma\), which is minimal for \(t : m\).

### 4.3 Examples

Our checker accepts a definition \(\lambda x = t\) if there exists a mode judgment \(\Gamma \vdash t : \text{Return}\) that assigns a mode to \(x\) in \(\Gamma\) that is Guard or smaller. The definition is rejected if the mode of \(x\) in the minimal judgment is Return or Dereference. Let us discuss various examples of definitions that are accepted or rejected, along with the corresponding minimal judgments.

*Separating Return from Guard, Delay.* The definitions \(\lambda x = \text{Fix } x\) or \(\lambda x = f : \text{Delay}\) are valid: the definitions admit the mode judgments \(x : \text{Guard} \vdash \text{Fix } x : \text{Return}\) and \(f : \text{Delay} \vdash \lambda x. f : \text{Return}\). On the other hand, the definition \(\lambda x = x\) is invalid, as the best judgment for its body is \(x : \text{Return} \vdash x : \text{Return}\), with \(x : \text{Return}\) (stricter than Guard) in \(\Gamma\).
**Separating Guard from Delay.** In the valid example \( \texttt{let rec } f = \lambda x. f \ x, \) the subterm \( f \ x \) is dereferencing \( f \); the mode of \( f \) in the outer environment is still Delay thanks to the composition \( \text{Delay}[\text{Dereference}] = \text{Delay} \). On the other hand, if we had moved the application outside the delay, \( \texttt{let rec } f = (\lambda x. f) \ u \), this definition would be rejected, as we would have \( f : \text{Dereference} \) thanks to the composition \( \text{Dereference}[\text{Delay}] = \text{Dereference} \).

Guard behaves differently than Delay: our constructors are strict, so dereferencing inside a guard is also a dereference. For example, considering a function \( g \) defined outside, both \( \texttt{let rec } x = g(\text{Fix } x) \) and \( \texttt{let rec } x = \text{Fix}(g \ x) \) are rejected, as the mode of \( x \) is Dereference in the right-hand side of the definition. In the first case this mode comes from the composition \( \text{Dereference}[\text{Guard}] = \text{Dereference} \), in the second case from \( \text{Guard}[\text{Dereference}] = \text{Dereference} \).

**Separating Delay from Ignore.** Notice that if we have \( x : \text{Delay} \vdash t[x] : \text{Return} \) then we have \( x : \text{Dereference}, g : \text{Dereference} \vdash g(t[x]) : \text{Return} \) but if we have \( x : \text{Ignore} \vdash t[x] : \text{Return} \) then we have \( x : \text{Ignore}, g : \text{Dereference} \vdash g(t[x]) : \text{Return} \). So a declaration of the form \( \texttt{let rec } x = g(t[x]) \) is rejected if \( t \) uses the variable \( x \) at mode Delay, but accepted if \( x \) is not used \((x : \text{Ignore})\).

**Separating Return from Dereference.** Similarly, if we have \( x : \text{Return} \vdash t[x] : \text{Return} \) (for example \( t[x] = x \) or \( \langle \text{let } y = x \text{ in } y \rangle \)), then we have \( x : \text{Guard} \vdash \text{Fix}(t[x]) : \text{Return} \), but if we have \( x : \text{Dereference} \vdash t[x] : \text{Return} \) then we have \( x : \text{Dereference} \text{ in the environment of } \text{Fix}(t[x]) \). In particular, \( \texttt{let rec } x = \text{Fix}(t[x]) \) is accepted in the first case and rejected in the second.

**Simple let examples.** (These examples are easier to follow by using the simple \( \texttt{let rec} \) rule than the general \( \texttt{let rec} \) rule.). What is the mode \( x \) in \( \texttt{let } z = \lambda y. \text{Pair}(x, y) \text{ in } \text{Fix } z? \) The mode of \( x \) in the definition \( \lambda y. \text{Pair}(x, y) \) is Delay, and the mode of \( z \) in the body \( \text{Fix } z \) is Guard. The final context (at global mode Return) is \( \text{Guard}[\text{Delay}] \), that is \( x : \text{Delay} \).

In the case of \( \texttt{let } z = \lambda y. \text{Pair}(x, y) \text{ in } g \ z \), the final context is \( \text{Dereference}[\text{Delay}] = x : \text{Dereference} \). Finally, the premise \( m' = \text{max}(m, \text{Guard}) \) of the rule comes into play when the body delays or ignores the defined variable: in the case of \( \texttt{let } z = g \ x \text{ in } y \), the mode of \( z \) in \( y \) is Ignore, but the mode of \( x \) in the whole term is not \( \text{Ignore}[\text{Dereference}] \), which would be \( x : \text{Ignore} \), but rather \( \text{max}(\text{Ignore}, \text{Guard})[\text{Dereference}] \), which is \( x : \text{Dereference} \).

**let rec examples.** The delicate aspect of the \( (x_i : \Gamma_i)^{\dag} + \text{rec } (x_i = t_i)^{\dag} \) judgment is the fixpoint of equations \( \Gamma' = \Gamma + \sum_{i \in I} \left( m_{i,j} \left[ \Gamma'_j \right] \right)^{\dag} \), which computes a “transitive closure” of usage modes of the mutually-recursively-defined variables. Consider for example the term \( t \) defined as

\[
\text{let rec } x' = x \text{ and } y = \text{Fix}(x') \text{ and } z = g \ y \text{ in } z
\]

We index the three definitions by \( j \in I \) with \( j \overset{\text{def}}{=} \{ x', y, z \} \). The contexts \( (\Gamma_i)^{\dag} \) of the rule correspond to the dependency of each right-hand-side on non-mutually-recursive variables. The \( (\Gamma'_i)^{\dag} \) are defined by a system of recursive equations, depending on each \( \Gamma_j \) and the mode of use of the variable \( j \) in the definition of \( i \). We have:

\[
\begin{align*}
\Gamma'_{x'} & \overset{\text{def}}{=} (x : \text{Return}) & \Gamma'_{x'} & \overset{\text{def}}{=} \Gamma_x \\
\Gamma'_{y} & \overset{\text{def}}{=} \emptyset & \Gamma'_{y} & \overset{\text{def}}{=} \Gamma_{y} + \text{Guard}[\Gamma'_{x'}] \\
\Gamma'_{z} & \overset{\text{def}}{=} (g : \text{Dereference}) & \Gamma'_{z} & \overset{\text{def}}{=} \Gamma_{z} + \text{Dereference}[\Gamma'_{y}]
\end{align*}
\]

The smallest fixpoint solution has \( \Gamma'_{x'} = (x : \text{Return}), \Gamma'_{y} = (x : \text{Guard}), \text{ and } \Gamma'_{z} = (g : \text{Dereference}, x : \text{Dereference}). \) In particular, notice how \( x \) is accessed at mode Dereference by \( z \), even though it does not syntactically appear in its definition. The whole term \( t \), in the ambient mode Return, is in the environment \( \Gamma'_{z} = (g : \text{Dereference}, x : \text{Dereference}) \). If we had used a simpler \( \texttt{rec} \) rule that would return the \( (\Gamma_i)^{\dag} \) instead of the \( (\Gamma'_i)^{\dag} \), immediate usage rather than
transitive usage, \( t \) would typed in the environment \( \Gamma_z = g : \text{Dereference} \), that is with \( x : \text{Ignore} \). This would be unsound, for example the vicious definition \( \text{let rec } x = t \) would be accepted.

4.4 Discussion

Declarative vs. algorithmic rules. A presentation of a type system can be more “algorithmic” or more “declarative”, sitting on a continuous spectrum. More-declarative systems have convenient typing rules corresponding to reasoning principles that are sound in the metatheory, but may be harder to implement in practice. More-algorithmic systems have more rigid inference rules, that are easier to implement as a checking or inference system (typically they may be syntax-directed); some valid reasoning principles may not be available as rules but only as admissible rules (requiring a global rewrite of the derivation) or not at all.

In our system, the variable and subsumption rules are typically declarative: the variable rule has an undetermined \( \Gamma \) context, and the subsumption rule makes the system non-syntax-directed. Without those two rules, it would not be possible to prove \( x : \text{Guard}, y : \text{Dereference} \vdash y : \text{Return} \), but only the stronger judgment \( x : \text{Ignore}, y : \text{Return} \vdash y : \text{Return} \).

On the other hand, our binding rule has an algorithmic flavor: it introduces a family of contexts \( (\Gamma_i')_i \) that is uniquely determined as the solution of a system of recursive equations

\[
\Gamma_i' = \Gamma_i + \sum (m_{i,j} \Gamma_j')_j,
\]

so its application requires computing a fixpoint. A more declarative presentation would allow “guessing” any family \( (\Gamma_i)_i \) that satisfies the inequations necessary for soundness:

\[
\begin{align*}
(\Gamma_i, (x_j : m_{i,j}) \vdash t_i : \text{Return})_i & \quad (m_{i,j} \leq \text{Guard})_j \\
(\Gamma_i \geq \Gamma_i + \sum (m_{i,j} \Gamma_j')_j) & \\
(\Gamma_i \geq \Gamma_i + \sum (m_{i,j} \Gamma_j')_j) & \\
(x_i : \Gamma_i)_i \vdash \text{rec } (x_i = t_i)_i
\end{align*}
\]

Backwards type systems. 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 mode system read “backwards” 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 reading type systems backwards can give a nice, compact, declarative presentation of certain demand analyses, in a language that type designers are already familiar with.

Our terminology follows the “backward analysis” notion described for logic programming languages by Genaim and Codish [2001], i.e. our algorithm answers the question (posed in that work) “Given a program and an assertion at a given program point, what are the weakest requirements on the inputs to the program which guarantee that the assertion will hold whenever execution reaches that point?” For our analysis, the assertion is the mode under which an expression is checked, and the requirements on the inputs correspond to the computed environment \( \Gamma \).

Backward analyses for functional languages also appear in work by Hughes [Hughes 1987]. A notable difference is that Hughes-style “demand analyses”\(^5\) are typically presented in a denotational style, using tools of domain theory. However, some recent work (such as the cardinality analysis by Sergey et al. [2017b]) presents backward analyses in a syntactic style more similar to that used in the present paper.

\(^5\)For a recent example, see the unpublished draft Sergey, Peyton-Jones, and Vytiniotis [2017a]. Thanks are due to Joachim Breitner for the reference.
Modes as modalities. Untyped or dynamically-typed languages can be seen as “uni-typed”, with a “universal type” $U$ of all values. Language constructions can be presented as section/retraction pairs from $U$ to a type that computes, such as $U \to U$ for functions or $U \times U \times \cdots \times U$ for tuples; for example, the untyped term $\langle x. t \rangle u$ can be explicitated into $\text{app}(\text{lam}(\langle x. t \rangle)) u$, for combinators $\text{app} : U \to (U \to U)$ and $\text{lam} : (U \to U) \to U$ such that $\text{app} \circ \text{lam}$ is the identity on $U \to U$ — but $\text{lam} \circ \text{app}$ gets stuck on non-functions.

Rather than types, it is more precise to see our modes as modalities on this universal type $U$. The hypothesis $x : m$ in a context would be a modal hypothesis $x :^m U$, and the section combinators are given modal types. Within the modal framework of Abel and Bernardy [2020] for example, writing $(m : A) \to B$ for the modal function type, some of our typing rules could be modelled with $\text{lam} : (\text{Delay} : (\text{Dereference} : U) \to U) \to U$, $\text{app} : (\text{Dereference} : U) \to (\text{Dereference} : U) \to U$, and for datatypes something like $\text{pack}^K_d : (\text{Guard} : U) \to U^d$, and $\text{unpack}^K_d : (\text{Dereference} : U) \to (U + U^d)$, where $K$ is a constructor of arity $d$, the $U+$ return value of unpacking indicates an incompatible constructor, and $U^d$ is $U \times U \times \ldots, d$ times.

This view naturally extends to supporting finer-grained analyses and abstraction of the sort found in other work (e.g. the system introduced by Dreyer [2004]). In our system every function argument has mode Dereference; a refinement that allowed types and modes to interact could support a range of modes for functions that used their arguments in different ways. However, this additional expressivity would require substantial changes to OCaml’s type system, and a proper treatment of abstraction would require some form of mode polymorphism. (For example, in the application function $\text{let } h \ g \ x = g \ x$, mode polymorphism is required to support applying the $h$ to all possible functions $g$, some of which dereference their argument, and some of which do not.)

Our aim of replacing OCaml’s existing syntactic check with a more principled version way did not justify this substantial additional complexity. More generally, our view is that abstraction over modes is more suited to module systems (for which the system described by Dreyer [2004] was developed), where types are already explicit, than to the term languages that our system is designed for. This view is supported by the fact that, as far as we know, although existing languages support a variety of checks on well-formedness of recursive definitions, there has been (as far as we know) no attempt in any of these to incorporate a system with support for mode abstraction.

5 META-THEORY: SOUNDNESS

5.1 Operational Semantics

Figure 4 presents our operational semantics, largely reused from Nordlander, Carlsson, and Gill [2008] with extensions (support for algebraic datatypes) and changes (use of reduction at a distance). Unless explicitly noted, the content and ideas in this Subsec 5.1 come from their work.

Weak values. As we have seen, constructors in recursive definitions can be used to construct cyclic values. For example, the definition $\text{let } \text{rec } x = \text{Cons} \ (\text{One} \ (\text{Empty})), x \rbrace$ 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 8 (Related Work)) defined semantics where $\beta$-redexes have the form $\langle \lambda x. t \rangle w$, 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 $\langle \lambda x. t \rangle v$. This is a difference from Nordlander, Carlsson, and Gill [2008]. On the other
Values \( \triangleright v := \lambda x . t | K(w_i)^j | L[v] \)

WeakValues \( \triangleright w := x, y, z | v | L[w] \)

ValueBindings \( \triangleright B ::= (x_i = v_j)^j \)

BindingCtx \( \triangleright L ::= \Box | \text{let rec } B \text{ in } L \)

\[
\begin{align*}
\text{EvalCtx} & \triangleright E ::= \Box | E[F] \\
\text{EvalFrame} & \triangleright F ::= \Box t | t \Box \\
& \quad | K((t_i)^i, \Box, (t_j)^j) \\
& \quad | \text{match } \Box \text{ with } h \\
& \quad | \text{let rec } b, x = \Box, b' \text{ in } u \\
& \quad | \text{let rec } B \text{ in } \Box \\
\end{align*}
\]

\[
\begin{align*}
\forall (K' x_j^j \rightarrow u') \in h, K \neq K' \\
& \quad \text{match } L[K(w_i)^j] \text{ with } (h | K(x_i)^i \rightarrow u | h') \rightarrow L[u[w_i/x_i]^i] \\
\end{align*}
\]

\[
\begin{align*}
L[\lambda x . t] v \rightarrow^\text{hd} L[t[v/x]] \\
\end{align*}
\]

\[
\begin{align*}
E[t] \rightarrow E[t'] \\
\quad (x = v)^\text{ctx} \in E \\
\quad (x = v)^\text{frame} \in E[F] \\
\quad (x = v)^\text{frame} \in \text{let rec } B \text{ in } \Box \\
\end{align*}
\]

\[
\begin{align*}
E[x] \rightarrow E[v] \\
\quad (x = v)^\text{ctx} \in E[F] \\
\quad (x = v)^\text{frame} \in \text{let rec } b, y = \Box, b' \text{ in } u \\
\end{align*}
\]

Fig. 4. Operational semantics

hand, we do allow cyclic datatype values by only requiring weak values under data constructors: the corresponding 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 \texttt{let rec}, ours included, is that \texttt{let rec }\( B \text{ in } \Box \) 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 \texttt{let x = v in u}, which is reduced to \( u[v/x] \) before going further. In \texttt{letrec} 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 \( t \rightarrow^\text{hd} t' \), 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\(^6\), 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. For example, the reduction sequence \( \texttt{let rec } B \text{ in } \lambda x . t \rightarrow^* \texttt{let rec } B \text{ in } t[u/x] \) is admissible.

\(^6\)See for example Accattoli and Kesner [2010], which links to earlier references on the technique.
in both systems, but the “useless” reduction \( \text{let rec } B \text{ in } x \rightarrow^* \text{let rec } B \text{ in } x \) is not present in our system. Reduction at a distance tends to make definitions crisper and simplify proofs.\(^7\)

**Reduction.** Reduction \( t \rightarrow t' \) may happen under any evaluation context. The first reduction rule is standard: any redex \( H[v] \) can be reduced under an evaluation context \( E \).

The second rule reduces a variable \( x \) 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 \((x = v) \in E\) to perform this lookup.

The lookup rule has worrying consequences for our rewriting relation: it makes it nondeterministic and non-terminating. Indeed, consider a weak value of the form \( K(x) \) used, for example, in a pattern-matching \( \text{match } K(x) \) 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 \( \text{match } K(K(x)) \) with \( h \), and we could keep looking indefinitely. Nordlander, Carlsson, and Gill [2008] discuss this in detail and prove that the reduction is in fact confluent modulo unfolding. (Allowing these irritating but innocuous behaviors is a large part of what makes their semantics simpler than previous presentations.)

**Example.** Consider the following program:

\[
\text{let rec } \infty = (\text{let rec } x = Sx \text{ in } x) \text{ in } \\
\text{match } \infty \text{ with } (Z \rightarrow \text{None} \mid S y \rightarrow \text{Some } y)
\]

The first binding \( \text{let rec } x = Sx \text{ in } x \) is not a value yet, only a weak value. The first reduction this program can take is to lookup the right-hand-side occurrence of \( x \):

\[
\rightarrow \text{let rec } \infty = (\text{let rec } x = Sx \text{ in } Sx) \text{ in } \\
\text{match } \infty \text{ with } (Z \rightarrow \text{None} \mid S y \rightarrow \text{Some } y)
\]

After this reduction \( \infty \), is bound to a value, so it can in turn be looked up in \( \text{match } \infty \):

\[
\rightarrow \text{let rec } \infty = (\text{let rec } x = Sx \text{ in } Sx) \text{ in } \\
\text{match } (\text{let rec } x = Sx \text{ in } Sx) \text{ with } \\
(Z \rightarrow \text{None} \mid S y \rightarrow \text{Some } y)
\]

At this point we have a match redex of the form \( \text{match } L[Sx] \), which gets reduced by lifting the binding context \( L \):

\[
\rightarrow \text{let rec } \infty = (\text{let rec } x = Sx \text{ in } Sx) \text{ in } \\
\text{let rec } x = Sx \text{ in } \text{Some } x
\]

### 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.

\(^{7}\) For an example of beneficial use of reduction-at-distance in previous work from the rewriting community, see the at-a-distance presentation of the \( \pi \)-calculus in Accattoli [2013].
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 $E[let \ rec \ (x = t, y = E') \ in \ u]$ for example, and that $x$’s binding has not yet been reduced to a value.

However, in the presence of data constructors that permit building cyclic values not all such cases are failures. For example the term $let \ rec \ x = Pair(x, t) \ in \ x$ 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 $(x = v) \in E$, unless $t$ is already a weak value.

To characterize failures during recursive evaluation, we propose to restrict ourselves to forcing contexts, denoted $E_f$, 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.

\[
\begin{align*}
\text{HeadFrame} & \ni H ::= \square \cdot v \mid \text{match} \square \text{with} \, h \\
\text{ForcingFrame} & \ni F_f ::= \square \cdot v \cdot \square \mid \text{match} \square \text{with} \, h \\
\text{ForcingCtx} & \ni E_f ::= L \mid E[F_f[L]]
\end{align*}
\]

Mismatch $\defeq \{E[H[v]] \mid H[v] \rightarrow^\text{hd}\}$

Vicious $\defeq \{E_f[x] \mid \not\exists v, (x = v) \in^\text{ctx} E_f\}$

Fig. 5. Failure terms

Mismatches are characterized by head frames, context fragments that would form a $\beta$-redex if filled with 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 $E_f$ takes into account the fact that recursive value bindings remain, floating around, in the evaluation context. A forcing frame $F_f$ 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 $E[F_f]$, but for example $F_f[let \ rec \ B \ in \ \square]$ 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 of these results are in Appendix B in the extended version.

**Lemma 1 (Forcing modes).** If $\Gamma, x : m \vdash E_f[x] : m$ with $m \geq \text{Return}$, then also $m_x \geq \text{Return}$.

**Theorem 2 (Vicious).** $\emptyset \vdash t : \text{Return}$ never holds for $t \in \text{Vicious}$.

**Theorem 3 (Subject reduction).** If $\Gamma \vdash t : m$ and $t \rightarrow t'$ then $\Gamma \vdash t' : m$.

**Corollary 1.** Return-typed programs cannot go vicious.
6 GLOBAL STORE SEMANTICS

In Section 5.1, we developed an operational semantics for \texttt{letrec} using explicit substitutions as “local stores” for recursive values. This technique was first studied in more theoretical research communities (pure lambda-calculus and rewriting), and imported in more applied programming-language works in the 1990s [Felleisen and Hieb 1992] and in particular local-store presentations of call-by-need and \texttt{letrec} [Ariola and Felleisen 1997].

Before local-store semantics were proposed, recursive values (and lazy thunks) were modelled using a “global store” semantics, more closely modelling the dummy-initialization-then-backpatching approach used in real-world implementations.

Explicit-substitution, local-store semantics enjoy at least the following advantages over the global-store semantics of \texttt{letrec}.

- Our explicit-substitution semantics is defined directly at the level of our term syntax, without going through additional features (global store, initialization and setting of memory locations) that are not part of the source language. Source-level semantics are typically more high-level and more declarative than lower-level semantics; they allow to reason on the source directly, without going through an encoding layer.

We can compare \texttt{letrec} to the problem of defining “tail calls”. One can teach tail-calls and reason about them through a compilation to a machine with an explicit call stack. But there is a source-level explanation of tail-calls, by reasoning on the size (and the growth) of the evaluation context; this explanation is simpler and makes tail-calls easier to reason about.

- Our local-store semantics allows more local reasoning; reducing a subterm in evaluation position only affects this subterm (which may contain explicit substitutions), instead of also affecting a global store. This makes it easier to define program-equivalence relations that are congruence (are stable under contexts), and generally to prove program-equivalence results.

Example 1 (Store duplication). For example, consider the two following programs, manipulating infinite lists of ones:

\begin{verbatim}
let o = (let rec x = Cons(1, x) in x) in f o o
f (let rec x = Cons(1, x) in x) (let rec x = Cons(1, x) in x)
\end{verbatim}

With our explicit-substitution semantics, it is trivial to show that these two terms are equivalent: they reduce to the same term, namely

\begin{verbatim}
f (let rec x = Cons(1, x) in Cons(1, x)) (let rec x = Cons(1, x) in Cons(1, x))
\end{verbatim}

(Note that \texttt{(let rec x = Cons(1, x) in x)} is not a value in our semantics, only a weak value; one needs to lookup \texttt{x} once to get a value so that the o-binding can be reduced.)

In a global-store semantics, on the contrary, it is not at all obvious that the two programs are equivalent; instead, they reduce to configurations of the following forms:

\begin{verbatim}
([x \mapsto Cons(1, x)], f x x)    ([x_1 \mapsto Cons(1, x_1), x_2 \mapsto Cons(1, x_2)], f x_1 x_2)
\end{verbatim}

These configurations are not equivalent by reduction; the equational theory would need a stronger equivalence principle that could de-duplicate bisimilar fragments of the store.

However, since a large part of our community is unfamiliar with local-store semantics, in the interest of accessibility, we also propose in this section a global-store semantics for our language. We show that the soundness result for our analysis can be lifted to this semantics: Return-typed programs cannot go vicious in the global-store semantics either. This is done by formalizing a compilation pass from the local-store language to the global-store+backpatching language, and
showing a backward simulation result — effectively redoing the compilation-correctness work of Hirschowitz, Leroy, and Wells [2003, 2009] in our setting.

6.1 Target Language

The global-store language is called the “target” language as we will prove the correctness of a compilation pass from the “source” (local-store) to “target” (global-store) languages. Its grammar and operational semantics are given in Figure 6.

A store location is created uninitialized with the \texttt{new} \(x\) \(t\) constructor, and can be defined exactly once by the assignment expression \(x \leftarrow t\); the heap binds locations to “heap blocks” that are either uninitialized (\(\bot\)) or a value. The write-once discipline is enforced by the \(x \leftarrow t\) reduction rule, which requires the heap block to be \(\bot\). Trying to read an uninitialized memory location is the dynamic error that our analysis prevents: it is the target equivalent of vicious terms, and we call it a “segfault” by analogy with the unpleasant consequences of the corresponding error in most compiled languages.

The reduction uses a distinguished data constructor, \texttt{Done}, playing the role of a unit value in our untyped semantics: it is the value returned by the evaluation of an assignment expression \(x \leftarrow v\).

**Remark 3 (Variables as locations).** We use variables as heap locations, an idea that goes back at least to [Launchbury 1993]. This is equivalent to having a distinguished syntactic category of locations thanks to the Barendregt convention — we work modulo \(\alpha\)-equivalence and can thus always assume that bound variables are distinct from free variables of the same term. Picking a “fresh enough” variable \(x\) each time we encounter a \texttt{new} binder gives a new memory location. In particular, those variables do not correspond to static binding positions in the program we started computing; each time a function is called, the \texttt{new} binders in its body bind over \(\alpha\)-equivalent fresh names. This technical choice will make it easier to relate to the source language, where recursive values are bound to variables in scope.

6.2 Parallel or Order-Independent Evaluation

In Section 6.1 we chose a distinguished \texttt{Done} constructor to represent a unit value in our target language. We now choose another distinguished constructor \(\texttt{Par}(t_1, \ldots, t_n)\) to represent a product of values to evaluate in an arbitrary order. We use a single constructor at arbitrary arities, but we could just as well use a family of constructors \(\texttt{Par}_n\).

**Definition 1 (Sequential and parallel evaluation).**

We define the following syntactic sugar:

\[
(t; u) \quad \text{def} = (\text{match } t \text{ with } (\text{Done} \rightarrow u))
\]

\[
\text{par} \left( t_i \right)_{i \in I} \quad \text{def} = (\text{match } \text{par} \left( t_i \right)_{i \in I} \text{ with } \text{par} \left( x_i \right)_{i \in I} \rightarrow \text{Done})
\]

The term \(\text{par} \left( t_i \right)_{i \in I}\) represents the evaluation of a family of \texttt{Done}-returning term, in an arbitrary order. In particular, \((\emptyset; u)\) and \(\text{par}(\left( t_i \right)_{i \in I}, \emptyset, \left( t_j \right)_{j \in J})\) are evaluation contexts, and the following reduction rules are derivable:

\[
(\emptyset, (\text{Done}; u)) \rightarrow^{\text{hd}} (\emptyset, u)
\]

\[
(\emptyset, \text{par}(\text{Done})_{i \in I}) \rightarrow^{\text{hd}} (\emptyset, \text{Done})
\]

6.3 Translation: Compiling \texttt{letrec} into Backpatching

We define in Figure 7 our translation \([t]\) from source to target terms, explaining recursive value bindings in terms of backpatching. The interesting case is \([\text{let rec } b \text{ in } u]\), the others are just a direct mapping on all subterms.

\[
[\text{let rec } (x_i = t_i)_{i \in I} \text{ in } u] \quad \text{def} = \text{new } (x_i)_{i \in I} \text{ in } \text{par} (x_i \leftarrow [t_i])_{i \in I}; [u]
\]

Then we compute the assignments $x \leftarrow t$.

$$TTerms \ni t, u ::= x, y, z$$
$$\quad \quad | \lambda x. t | t u$$
$$\quad \quad | K(t_i)^I | \text{match } t \text{ with } h$$
$$\quad \quad | \text{new } x \text{ in } t$$

$$\text{Handlers, patterns: as in the source language}$$

$$TValues \ni v ::= \lambda x. t | K(w_i)^I$$

$$TWeakValues \ni w ::= x, y, z | v$$

$$\text{ValueHeaps } \ni B ::= \emptyset | B[x \mapsto v]$$

$$\text{Heaps } \ni H ::= \emptyset | H[x \mapsto v?]$$

$$\text{HeapBlocks } \ni v? ::= v | \bot$$

$$TEvalCtx \ni E ::= \square | E[F]$$

$$TEvalFrame \ni F ::= \square t \mid t \square$$

$$\text{TFEvalCtx } \ni E_f ::= \square | E[F_f]$$

$$\text{TFForcingFrame } \ni F_f ::= \square t \mid t \square$$

$$\text{Fig. 6. Syntax and reduction of the global-store (target) language}$$

$$\text{Fig. 7. Compiling } \text{let rec } (x_i = t_i)^I \text{ in } u, \text{ we create uninitialized store cells for each } x_i^8, \text{ then we compute the assignments } x_i \leftarrow \llbracket t_i \rrbracket \text{ in an arbitrary order, and finally we evaluate } \llbracket u \rrbracket. \text{ Note that all the } x_i \text{ are in the scope of each } \llbracket t_i \rrbracket; \text{ the translation respects the scoping of the let rec } (x_i = t_i)^I \text{ construct.}$$

$$\llbracket x \rrbracket \overset{\text{def}}{=} x$$

$$\llbracket \lambda x. t \rrbracket \overset{\text{def}}{=} \lambda x. \llbracket t \rrbracket$$

$$\llbracket K(t_i)^I \rrbracket \overset{\text{def}}{=} K\left(\llbracket t_i \rrbracket^I\right)$$

$$\llbracket \text{match } t \text{ with } (p_i \rightarrow t_i)^I \rrbracket \overset{\text{def}}{=} \text{match } \llbracket t \rrbracket \text{ with } (p_i \rightarrow \llbracket t_i \rrbracket)^I$$

$$\llbracket \text{let rec } (x_i = t_i)^I \text{ in } u \rrbracket \overset{\text{def}}{=} \text{new } (x_i)^I \text{ in } \text{par}(x_i \leftarrow \llbracket t_i \rrbracket)^I; \llbracket u \rrbracket$$

$$^8(\text{new } (x_i)^I \text{ in } \square) \text{ denotes a sequence of } (\text{new } x_i \text{ in } \square) \text{ binders in an arbitrary order. In particular, } (\text{new } \emptyset \text{ in } t) \text{ is just } t.$$
6.4 Relating Target Terms back to Source Terms

**Simple rules**

\[
\begin{align*}
\lambda x.t & \sim (\emptyset, \lambda x.t) & t \sim (H_t, t') & u \sim (H_u, u') \\
(\tau \sim (H_{\tau}, \tau')) & \sim (H_{\tau}, \tau') & u \sim (H_u \uplus H_{\tau}, u' \tau') & t \sim (H, t') \\
\end{align*}
\]

**letrec rules**

\[
\begin{align*}
\text{INIT} & \quad J \neq \emptyset \\
& \quad \text{let rec } (x_i = t_i)^{i \in I}, (x_j = t_j)^{j \in J} \text{ in } u \\
& \quad \sim ([x_i \mapsto \bot]^{i \in I}, \text{new } (x_j)^{j \in J} \text{ in } \text{par}(x_k \leftarrow t_k)^{k \in J} ; \llbracket u \rrbracket) \\
\text{WRITE} & \quad (v_{s,i} \sim (B_i, v_{t,i}))^{i \in I}, (t_{s,j} \sim (H_j, t_{t,j}))^{j \in J} \\
& \quad \text{let rec } (x_i = v_{s,i})^{i \in I}, (y_j = t_{s,j})^{j \in J} \text{ in } u \\
& \quad \sim ([x_i \mapsto v_{t,i}]^{i \in I} [y_j \mapsto \bot]^{j \in J} \uplus (B_i)^{i \in I} \uplus (H_j)^{j \in J} \uplus \text{par}(\text{Done})^{i \in I}, (y_j \leftarrow t_{t,j})^{j \in J} ; \llbracket u \rrbracket) \\
\text{DONE} & \quad (v_{s,i} \sim (B_i, v_{t,i}))^{i \in I} \\
& \quad \text{let rec } (x_i = v_{s,i})^{i \in I} \text{ in } u \sim ([x_i \mapsto v_{t,i}]^{i \in I} \uplus (B_i)^{i \in I} \uplus H, \text{Done} ; \llbracket u \rrbracket) \\
\text{FURTHER} & \quad (v_{s,i} \sim (B_i, v_{t,i}))^{i \in I} \quad u_s \sim (H, u_t) \\
& \quad \text{let rec } (x_i = v_{s,i})^{i \in I} \text{ in } u_s \sim ([x_i \mapsto v_{t,i}]^{i \in I} \uplus (B_i)^{i \in I} \uplus H, u_t) \\
\end{align*}
\]

**Heap rules**

\[
\begin{align*}
\text{HEAP-WEAKEN} & \quad t_s \sim (H, t_t) \quad \forall x \in \text{dom}(B), x \notin (H, t_t) \\
& \quad t_s \sim (H \uplus B, t_t) \\
\text{HEAP-COPY} & \quad \phi \text{ compatible with } H \\
& \quad t_s \sim (H, t_t) \quad \text{dom}(\phi) \subseteq \text{dom}(H) \\
& \quad t_s \sim (\phi(H), \phi(t_t)) \\
\end{align*}
\]

Fig. 8. Head term relation \( t \sim (H, t') \)

We want to prove that this translation scheme is safe; that source terms that do not go vicious translate into target terms that do not segfault. This requires a backward simulation property: any reduction path from a translation in the target (in particular, a reduction path that leads to a segfault) needs to be simulated by a reduction path on the original source term (in particular, a reduction path that goes vicious).

We define a head term relation \( t \sim (H, t') \) that relates the parts of a term and a configuration that are in reducible position. For non-reducible positions, we use the direct embedding \( \llbracket t \rrbracket \) above. In
this relation, \( H \) contains not all the locations that are bound in \( t' \), but only those that correspond to \texttt{letrec}-binding found in \( t \); intuitively, \( H \) is the disjoint union of all the \texttt{letrec}-bindings in \( t \), seen as local store fragments.

The rules for the \texttt{letrec} constructs correspond to a decomposition of the various intermediate states of the reduction of their backpatching compilation.

Finally, the “heap rules” give reasoning principles to bridge the difference between the way the local and global stores evolve during reduction. In our source-level semantics, explicit substitutions (local store) may be duplicated or erased during reductions involving the values they belong to. Store duplication is expressed by a variable-renaming substitution \( \phi \), that “merges” different fragments of the global store together; the side-condition \((\phi \text{ compatible with } H)\) guarantees that the resulting store is well-formed.

For reasons of space, we moved the explanation of this relation, including the definition of \((\phi \text{ compatible with } H)\), as well as the proofs of its properties, to Appendix C (Compilation to Global Store: Simulation Proof) in the extended version. The two key results are mentioned here, guaranteeing that our analysis is also sound for this global-store semantics.

**Theorem 4 (Backward Simulation).**
If
\[
t_s \sim (H, t_t) \quad \quad (H, t_t) \rightarrow (H', t'_t)
\]
then \( \exists t'_s, \)
\[
t_s \xrightarrow{\cdot} t'_s \quad \quad t'_s \sim (H', t'_t)
\]

**Theorem 5 (Return-typed programs cannot segfault).**
\[
\emptyset \vdash t_s : \text{Return} \quad \quad \wedge \quad \quad (\emptyset, [t_s]) \xrightarrow{*} (H, t'_t) \quad \quad \implies \quad \quad (H, t'_t) \notin \text{Segfault}
\]

Theorem 5 (Return-typed programs cannot segfault) guarantees that our analysis is sound for both our source language and its backpatching translation. Theorem 4 (Backward Simulation) also tells us that our source semantics has “enough” reduction rules compared to the global-store semantics. For example, if the global-store semantics computes a value for a term, then the source semantics would have computed a related value.

**7 EXTENSION TO A FULL LANGUAGE**

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

**7.1 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; e.g. variant types may contain constructors with different arities, resulting in different in-memory sizes, 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,

\[
\text{if } p \text{ then } (\text{fun } x \to x) \text{ else } (\text{fun } x \to \text{not } x)
\]

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

\[
\text{if } p \text{ then } (\text{fun } x \to x + 1) \text{ else } (\text{fun } x \to x + \text{offset})
\]

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 statically resolvable to a global name.)

Relation to the mode system. The mode system corresponds to a correctness criterion on the operational semantics of programs; it is independent of compilation schemes. In contrast, the size discipline corresponds to a restrictive compilation strategy for value recursion that involves rejecting certain definitions. The size discipline is formalized by Hirschowitz et al. [2009]; it would be possible to incorporate it into our system, modelling it as a separate judgment to be checked for well-moded definitions (rather than as an enrichment of the mode judgment). However, the resulting system would be less portable to programming languages whose value representations differ from OCaml’s, and which consequently would not use the same size discipline.

7.2 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 10.5) and y = 1.5
```

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.

7.3 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 \( M.\text{Exit} 42 \) may
dereference the module \( M \) where \( \text{Exit} \) 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:

```ocaml
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 gener-
ated 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.)

### 7.4 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

```ocaml
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:

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

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

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

### 7.5 Laziness

OCaml’s evaluation is eager by default, but it supports an explicit form of lazy evaluation: the
programmer can write \textbf{lazy} \( e \) and \textbf{force} \( e \) to delay and force the evaluation of an expression.

The OCaml implementation performs a number of optimizations involving \textbf{lazy}. For example,
when the argument of \textbf{lazy} is a trivial syntactic value (variable or constant) for which eager and
lazy evaluation behave equivalently, the compiler eliminates the allocation of the lazy thunk.

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

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

while the following definition is allowed by the check, since the argument to \textbf{lazy} is not sufficiently
trivial to be subject to the optimization:
\[\text{let rec } x = \text{lazy } (y + 0) \text{ and } y = \ldots\]

Our typing rule for \text{lazy} takes this into account: “trivial” thunks are checked in mode Return rather than Delay.

## 8 RELATED WORK

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 \(\tau^0 \rightarrow \tau'\) accesses its argument to return a result, while a \(\tau^1 \rightarrow \tau'\) 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 \text{let rec } obj = \text{class_constructor } obj\ param\ s, arising from object-oriented encodings, where \text{class_constructor} has a type \(\tau^0 \rightarrow \ldots\).

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 \text{let rec } obj = \text{constr } obj\ param\ s 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 \(\tau^\alpha \rightarrow \tau'\), 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 is 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. Their work is an inspiration for our own compilation result: they provide a source-level semantics based on floating bindings upwards in the term (similar to explicit substitutions or local thunk stores), and prove correctness of compilation to a global store with backpatching.

Our source-level semantics is close to theirs in spirit (we would argue that the use of reduction at a distance is an improvement), and our compilation scheme and its correctness proof are not novel compared to their work — they are there to provide additional intuition. The main contribution of our work is our mode system for recursive declarations, which is expressive enough to capture OCaml value definitions, yet simple and easy to infer.

A natural question for our work is whether the access-mode derivations we build in our safety check can inform the compilation strategy for recursive values. We concentrate on safety, but there is ongoing work by others on the compilation method for recursive values, which partly goes in this direction.

Fixing Letrec (Reloaded). Fixing Letrec (Reloaded) [Ghuloum and Dybvig 2009; Waddell, Sarkar, and Dybving 2005] is a nice brand of work from the Scheme community, centered on producing efficient code for recursive value declarations, even in presence of dynamic checks for the absence of uninitialized-name reads. It presents a static analysis, both for optimization purposes (eliding dynamic safety checks) and user convenience. The analysis is described by informal prose, but it is
similar in spirit to our mode analysis (using modes named “protected”, which corresponds to our Delay, “protectable” which sounds like Return and “unsafe” which is Dereference). We precisely describe an analysis (richer, as it also has a Guard mode) and prove its correctness with respect to a dynamic semantics.

**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 := τ = 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 : τ at the type `box(X, τ)` instead of τ (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 τ to τ’, 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 \( \forall X. box(X, τ) \rightarrow τ’ \) instead of the simple function type \( τ \rightarrow τ’ \). (It is possible to inject τ into `box(X, τ)`, 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 \( ∀α.α \rightarrow List(α) \rightarrow List(α) \) to the box-expecting type \( ∀α.α \rightarrow ∀X. box(X, List(α)) \rightarrow List(α) \).

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. However, we believe that any well-typed program that is accepted by our mode system could be encoded in Dreyer’s system, roughly as follows:

- if in the context \( \Gamma \) of a derivation \( \Gamma \vdash t : \text{Return} \) in our system, we have \( x : \text{Dereference} \), then in the encoding the corresponding effect variable X would be an ambient capability \( (\Gamma \vdash t : τ|T) \) with \( X ∈ T \)
- on the other hand, if we have \( x : \text{Return} \) or a more permissive mode, then we would give the corresponding term variable in the encoding type `box(X, T)`

So, for example, \( x : \text{Dereference} \vdash t + 1 : \text{Return} \) would be encoded as \( x, x + 1 : \text{Int}[X] \) but \( x : \text{Guard} \vdash \{ t = x \} : \text{Return} \) would be encoded as \( x, x : \text{box}(X, τ) \vdash \{ t = x \} : \{ t : τ \}[\emptyset] \).

The whole derivation of an encoding of a valid recursive definition would have a non-boxed type on the right-hand side, without any of the effect variables of the recursively-defined variable in the ambient context.

**Strictness analysis.** Our analysis can be interpreted as a form of strictness or demand analysis, with modes above Return being non-strict and Dereference being the forcing mode. Note however that we use a may-analysis (a Dereference variable may be dereferenced) while strictness optimizations usually rely on a must-analysis (we only give Forcing when we know for sure that forcing happens); to do this one should change our interpretation of unknown functions to be conservative in the other direction, with mode Ignore rather than Dereference for their arguments. More importantly, strictness analyses typically try to compute more information than our modes: besides the question of whether a given subterm will be forced or not, they keep track of which prefixes of the possible
term shapes will get forced — this is more related to the finer-grained spaces of “ranks” or “degrees” for recursive functors.

**Graph typing.** Hirschowitz also collaborated on static analyses for recursive definitions in Bardou [2005]; Hirschowitz and Lenglet [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 Dereference mode. 0 conflates our Guard and 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 Delay mode, and finally $+\infty$ is our 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.

One contribution of our work is to show that a more standard syntactic approach can replace the graph representation. Note that our typing rule for mutual-recursion uses a fixpoint computation, reminiscent of their transitive-closure computation but within a familiar type-system presentation.

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 mode 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 7.1 (The Size Discipline).

F$\sharp$. Syme [2006] proposes a simple translation of mutually-recursive definitions into delay and force constructions that introduce and eliminate lazy values. For example, let rec $x = t$ and $y = u$ is turned into the following:

```ocaml
let rec $x_{\text{thunk}} = \text{lazy (}[\text{force } x_{\text{thunk}}/x, \text{force } y_{\text{thunk}}/y])$
and $y_{\text{thunk}} = \text{lazy (}[u [\text{force } x_{\text{thunk}}/x, \text{force } y_{\text{thunk}}/y])}$
let $x = \text{force } x_{\text{thunk}}$ and $y = \text{force } y_{\text{thunk}}$
```

With this semantics, evaluation happens on-demand, which the recursive definitions evaluated at the same 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:

```ocaml
let rec $\text{ones}_{\text{thunk}} = \text{lazy (}[1 [\text{force } \text{ones}_{\text{thunk}}])]$
```

Furthermore, the translation affects the semantics of programs in surprising ways: in particular, the implicit introduction of laziness into definitions that start new threads can lead to unexpected multiple execution of computations and to race conditions.

Nowadays, F$\sharp$ 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 ignoring occurrences within delaying terms (function abstractions or objects or lazy thunks) even if those delaying terms may themselves be used
(i.e. respectively called or accessed or 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.

*Needed computations.* Further connections between laziness and recursive call-by-value definitions may be drawn: for example, Chang and Felleisen [2012] characterize call-by-need by introducing needed computations, which are similar in spirit to our idea of forcing contexts. However, the set of computations characterized by the two ideas are different in practice: for example, in our system $f \ (\text{fst}\_\-)\_\text{is a forcing context, but the hole is not in “needed position” for call-by-need.}$

Intuitively, needed computations correspond to positions at which any choice of reduction order must force the computation to make progress, while forcing contexts correspond to positions where some choice of reduction order may force the computation (and fail if the value is initialized).

*Operational semantics.* Felleisen and Hieb [1992] and Ariola and Felleisen [1997] propose local-store semantics (for a call-by-value store and a call-by-need thunk store, respectively) that can express recursive bindings. The source-level operational semantics of Hirschowitz, Leroy, and Wells [2003, 2009] is more tailored to recursive bindings, manipulated as explicit substitutions, although the relation to standard explicit-substitution calculi is not made explicitly. They also provide a global-store 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.

Our own experience presenting this work is that local-store semantics has been largely forgotten by the programming-language community, which is a shame as it provides a better treatment of recursive definitions (or call-by-need) than global-store semantics.

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 [Nordlander, Carlsson, and Gill 2008; Syme 2006] 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 an uninitialized variable is a stuck term whose evaluation results in a fatal error; 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.

## 9 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 also 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.
REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. In ICFP.


