A practical type system for generalized recursion

Tom Hirschowitz, Serguei Lenglet

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

A practical type system for generalized recursion

Tom Hirschowitz and Sergueï Lenglet

May 2005

Research Report N° 2005-22

École Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37 Fax : +33(0)4.72.72.80.80 Adresse électronique : `lip@ens-lyon.fr`

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

INRIA

A practical type system for generalized recursion

Tom Hirschowitz and Sergueï Lenglet

May 2005

Abstract

The ML language is equipped with a sophisticated module system, especially thanks to its notions of functor (higher-order functions on modules) and of controlled type abstraction (opaque or transparent types).
Nevertheless, an important weakness of this system hinders modularization: the impossibility to define mutually recursive modules. In particular, mutually recursive functions must all reside in the same module.
Recently, Leroy extended the OCaml language, a dialect of ML, with an unsafe notion of recursive modules. In this extension, one can define recursive modules, but the system does not check that they are well-founded. If not, the system throws an exception at runtime, which is annoying given the strong typing of ML.
Powerful type systems have been proposed to tackle this issue, but they require rather deep modifications to the ML type system.
This report presents a system requiring only local modifications to the ML type system. We prove its soundness by injection into one of the evoked more general formalisms.

Keywords: Programming languages, semantics, typing, modularity, recursion.

Résumé

Le langage ML est doté d'un système de modules sophistiqué, notamment grâce aux foncteurs (fonctions d'ordre supérieur sur les modules) et à son mécanisme d'abstraction de types contrôlée (types manifestes ou abstraits).
Cependant, une faiblesse importante de ce système gêne la modularisation des programmes: l'impossibilité de définir des modules de façon mutuellement récursive. Notamment, les définitions de fonctions mutuellement récursives doivent toutes résider dans le même module.
Récemment, Leroy a étendu le langage OCaml, un dialecte de ML, avec une notion non sûre de modules récursifs. Avec cette extension, on peut définir des modules récursifs, mais le système ne vérifie pas que ces définitions sont bien fondées. Lorsqu'elles ne le sont pas, le système lance une exception à l'exécution, ce qui cadre mal avec le typage fort de ML.
Des systèmes de types assez généraux ont été proposés récemment pour gérer ce problème, mais leur mise en œuvre demande une modification en profondeur du typage de ML.
Ce rapport propose un système ne nécessitant que des modifications locales au typage de ML. Nous prouvons sa sûreté par injection dans l'un des formalismes plus généraux évoqués ci-dessus.

Mots-clés: Langages de programmation, sémantique, typage, modularité, récursion.

Contents

1 Introduction

The ML language [19] is equipped with a sophisticated module system [18, 14, 8, 5], especially thanks to its notions of functor (higher-order functions on modules) and of type abstraction (opaque or transparent types). Nevertheless, an important weakness of this system hinders modularization: the impossibility to define mutually recursive modules: mutually recursive functions must all reside in the same module.

Recently, Leroy extended the OCaml language [17], a dialect of ML, with an unsafe notion of recursive modules. In this extension, one can define recursive modules, but the system does not check that they are well-founded. If they are not, the system eventually throws an exception at runtime, which weakens ML typing. Powerful type systems have been proposed to tackle this issue [9, 7], but they require rather deep modifications to the ML type system, which are particularly annoying at the level of types, where they introduce new, complex forms of functor types. This paper presents a system requiring only local modifications to the ML type system. We prove its soundness by injection into a more powerful type system. The latter slightly improves over the one proposed in Hirschowitz's PhD thesis [9].

Instead of trying to define the most powerful notion of recursion, we only target the common patterns of recursive module programming, which include basic modules, of the shape `struct ... end`, but also functor applications. A well-known example using functor applications is an implementation of Okasaki's bootstrapped heaps [20] with recursive modules by Dreyer etal. [6]. In Fig. 1, we consider a simpler, often wished-for example [15]. Such definitions are difficult to type-check in a separate compilation setting. Indeed, their safety depends on the body of the functor `Set.Make` and in particular on the way it uses the argument variable, which is impossible to encode in its signature. Our solution relies on two main technical devices:

Refined functor types We distinguish functors types according to the way their bodies depend on their arguments. Thus, `Set.Make` has a weak functor type (whose arrow is written $\sigma_1 \Rightarrow \sigma_2$ instead of $\sigma_1 \rightarrow \sigma_2$. This makes the necessary information available during type-checking.

Weak and strong dependencies We distinguish two kinds of dependencies, strong and weak. Intuitively, the dependency of a module expression e on a variable x is *weak* only if

1. the value denoted by x is represented by a heap block, and
2. the evaluation of e does not lead to inspect the contents of this heap block.

Since we are designing a static dependency analysis, we use an approximation of this notion.

The formal setting of our study is the $\lambda_\circ$-calculus, a λ-calculus with a very powerful `let rec` construct, which has been shown to be efficiently compilable [11]. This calculus adequately models ML recursive modules, and allows to abstract over the complex issues of typing recursive modules w.r.t. type components [6]. In its experimental extension of OCaml with recursive modules, Leroy implemented a type-checking algorithm described in an informal note [15]. The discussion and formalization of this algorithm are beyond the scope of this paper.

The paper is organized as follows. In Sect. 3, we present $\lambda_\circ$ and its operational semantics, along with some examples illustrating how the language models recursive modules. In Sect. 6, we present our type system for $\lambda_\circ$, which relies on an abstract notion of *degree* to model the dependencies of modules on their free variables. This section also explains why the soundness of the type system is difficult to prove directly. Section 5 introduces a more powerful and more complex type system, whose soundness can be proved directly (see appendices A and B). After stating the soundness theorem, we prove the soundness of the simple type system of Sect. 6 by direct injection into the complex one. We do not examine related work in this research report.

```
module A : sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end = struct
   type t = Leaf of string | Node of ASet.t
   let compare t1 t2 = match (t1, t2) with
   | (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
   | (Leaf _, Node _) -> 1
   | (Node _, Leaf _) -> -1
   | (Node n1, Node n2) -> ASet.compare n1 n2
end
and ASet : Set.S with type elt = A.t = Set.Make(A)
```

Figure 1: Recursive functor application in OCaml

2 Motivation for $\lambda_\circ$: immediate in-place update

The particular choice of $\lambda_\circ$ as a model for recursive modules has to be motivated. We do this in the present section, explaining its definition in the light of the compilation scheme used for recursive modules in OCaml. For compiling recursive modules, several known methods apply. We successively review them and explain how this leads to $\lambda_\circ$.

2.1 Lazy evaluation

A very flexible way of compiling recursive modules is to use lazy evaluation, which allows to compile any definition. The idea is that recursive modules are first defined as *thunks*, and then their evaluation is forced. This raises an exception at execution time in case the definition is ill-founded.

For example, in OCaml syntax, a definition of the shape

```
module rec A : T = e
```

is compiled to

```
let rec A = lazy e in
let A = Lazy.force A
```

As long as e does not force the evaluation of `A`, the evaluation goes well, thus allowing to implement the example in Fig. 1. Nevertheless, lazy evaluation entails runtime tests and indirections, which make it a rather inefficient method. Moreover, it weakens the strong typing of ML.

2.2 Backpatching

Another method that also allows to compile any definition is to use a *backpatching* semantics, as done in the Scheme language [13]. The idea consists in first assigning a reference cell to each recursive variable, initialized with some dummy value (denoted by `nil` in the following). Then, the right-hand sides are evaluated. Until this point, any attempt to dereference the cells is a run-time error. Finally, the reference cells are updated with the obtained values, and the definitions can be considered fully evaluated.

The backpatching scheme leaves some flexibility as to when the reference cells bound to recursively-defined variables are dereferenced. In Scheme, every occurrence of these variables in the lexical scope of the `letrec` binding causes an immediate dereference. In Boudol's compilation scheme for the λ_{B} intermediate language [3], the dereferencing is further delayed because arguments to functions are passed by reference rather than by value. The difference is best illustrated on the definition $x = (\lambda y.\lambda z.\mathtt{if}\ z = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y\ (z-1))\ x$. In Scheme, this definition compiles down to the following intermediate code

$$\begin{array}{l}\texttt{let}\, x = \texttt{ref nil in} \\ x := (\lambda y.\lambda z.\texttt{if}\ \ z = 0\ \texttt{then}\ 1\ \texttt{else}\ \ y\ (z-1))\ !x\end{array}$$

and therefore fails at run-time because the reference x is accessed at a time when it still contains `nil`. In Boudol's compilation scheme, the y parameter is passed by reference, resulting in the following compiled code:

$$\begin{array}{l}\texttt{let}\, x = \texttt{ref nil in} \\ x := (\lambda y.\lambda z.\texttt{if}\ \ z = 0\ \texttt{then}\ 1\ \texttt{else}\ !y\ (z-1))\ x\end{array}$$

Here, x is passed as a function argument without being dereferenced, therefore ensuring that the recursive definition evaluates correctly. The downside is that the recursive call to y has now to be preceded by a dereferencing of y.

Using the second solution, it is possible to implement the example in Fig. 1. However, in both cases, a drawback of the backpatching approach is that recursive calls to a recursively-defined function must go through one additional indirection. For well-founded definitions, this indirection seems superfluous, since no further update of the reference cells is needed. In practice, compilers recognize and optimize some common kinds of recursive definitions, typically functions, but it appears preferable to rely on a more general method.

2.3 In-place update

The *in-place update* scheme [4] is a variant of the backpatching implementation of recursive definitions that avoids the additional indirection just mentioned. It is used in the OCaml compilers [16].

The in-place update scheme implements `let rec` definitions that satisfy the following two conditions. For any mutually recursive definition $x_1 = e_1 \dots x_n = e_n$, first, the value of each definition should be represented at run-time by a heap allocated block of statically predictable size; second, for each i, the computation of e_i should not need the value of any of the definitions e_j, but only their names x_j. As an example of the second condition, the recursive definition $f = \lambda x.(...f...)$ is accepted, since the computation of the right-hand side does not need the value of f. We say that it *weakly* depends on f. In contrast, the recursive definition $f = (f\ 0)$ is rejected. We say that the right-hand side *strongly* depends on f.

The evaluation of a `let rec` definition with in-place update consists of three steps. First, for each definition, allocate an uninitialized block of the expected size, and bind it to the recursively-defined identifier. Those blocks are called *dummy* blocks. Second, compute the right-hand sides of the definitions. Recursively-defined identifiers thus refer to the corresponding dummy blocks. Owing to the second condition, no attempt is made to access the contents of the dummy blocks. This step leads, for each definition, to a block of the expected size. Third, the contents of the obtained blocks are copied to the dummy blocks, updating them in place. One could argue that the obtained values could directly fill the dummy blocks. However, this would require a special evaluation scheme, whereas here, they are evaluated just like any other expression.

For example, consider a mutually recursive definition $x_1 = e_1, x_2 = e_2$, where it is statically predictable that the values of the expressions e_1 and e_2 will be represented at runtime by heap allocated blocks of sizes 2 and 1, respectively. Here is what the compiled code does, as depicted in Fig. 2. First, it allocates two uninitialized heap blocks, at addresses ℓ_1 and ℓ_2, of respective sizes 2 and 1. This is called the pre-allocation step. As a second step, it computes e_1, where x_1 and x_2 are bound to ℓ_1 and ℓ_2, respectively. The result is a heap block of size 2, with possible references to the two uninitialized blocks. The same process is carried on for e_2, resulting in a heap block of size 1. The third and final step copies the contents of the two obtained blocks to the two uninitialized blocks. The result is that the two initially dummy blocks now contain the proper cyclic data structures, without the indirection inherent to the backpatching semantics.

2.4 Relaxed in-place udpate

In spite of its advantages, the in-place update scheme cannot be used directly for compiling recursive modules. Indeed, the conditions for it to be sound are impossible to check syntactically. We have

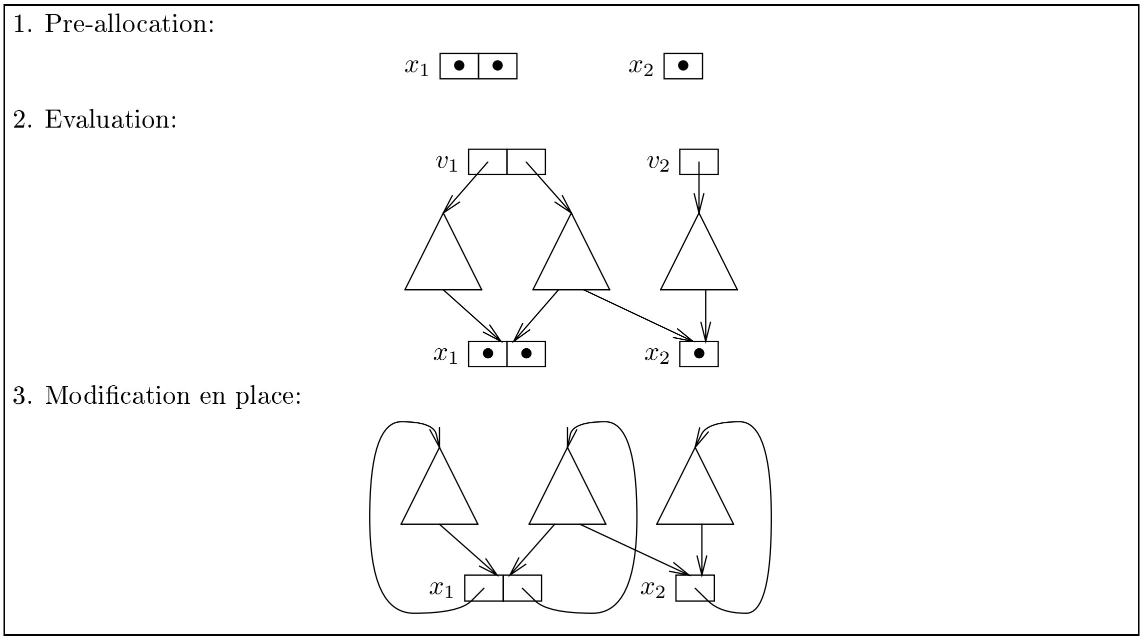


Figure 2: The in-place update scheme

seen in Sect. 1 that some functor applications are highly desirable as right-hand sides of recursive module definitions. However, recursive modules are intended to remain compatible with separate compilation, so it is impossible to check that a functor application will not dereference its argument without knowing the body of the functor.

This is why Leroy implemented another solution [15], which he calls *relaxed in-place update*. It is a variant of the in-place udpate scheme, which prevents segmentation faults by initializing the dummy blocks with *sound* values. A sound value is a value that can be initialized with a value of the same type, which when used raises the exception `Undefined_recursive_module`. A sound module is defined has having only sound components.

The sound values are those of either functional or lazy type, which we can initialize with `fun x -> raise Undefined_recursive_module` and `lazy (raise Undefined_recursive_module)`, respectively.

Furthermore, it is insufficient to simply apply the in-place update scheme with additional initialization of sound values. Indeed, this requires that all the recursively-defined modules contain only sound values, and the example in Fig. 1 contains the module `ASet`, which has a `empty` component of type `ASet.t`. In order to solve this problem, Leroy refines the in-place update scheme as follows. In order to compile a recursive definition of the shape `module rec` X_1`:` M_1`=` m_1`and ...` X_n`:` M_n`=` m_n, the compiler first separates sound modules from unsound ones, and rejects programs containing forward references to unsound modules, in the following sense.

Definition 1 (Forward reference)
A forward reference *is a pair* (i, j) *such that* $i \leq j$*, and* m_i *mentions* x_j*.*

Then, all the sound modules X_i are bound to blocks initialized with sound values of the expected type. Then, the definitions $X_i = m_i$ are successively evaluated, and *immediately* updated in place if necessary: if m_i is unsafe, then it is simply `let` bound to X_i; otherwise it is used to update X_i in place.

Remark 1 (Reordering)
In fact, if possible, the compiler reorders the definitions to prevent forward references to unsound modules. In the following, we consider this step as preprocessing.

Let us see how the examples of Sect. 1 are compiled. In the example in Fig. 1, the module `A` is

```
module type T = sig
  val f : int -> int
end

module rec A : T = B
and B : T = struct
  let f x = x
end

A.f 0;;
```

Figure 3: Another recursive modules example in OCaml

sound, since `compare` has a functional type, and `ASet` is unsafe. After reordering, `ASet` is evaluated first. The generated code is

```
(* Step 1 : allocation and initialization *)
let A = { compare = fun x -> raise (Undefined_recursive_module) }
    in
(* Step 2 : evaluation *)
let ASet = Set.Make (A) in
update (A, {compare = fun x y -> match (x, y) with ...});
...
```

The exception`Undefined_recursive_module` is not raised, because the functor `Set.Make` does not access `A.compare`.

Consider now the code in Fig. 3. The call to `A.f` throws the `Undefined_recursive_module` exception. Indeed, here is how this code is compiled: `A` and `B` only contain compoennts of functional type, so they are both sound. The module `A` is evaluated first. The generated code is

```
(* Step 1 : allocation and initialization *)
let A = { f = fun x -> raise (Undefined_recursive_module) }
and B = { f = fun x -> raise (Undefined_recursive_module) } in
(* Step 2 : evaluation *)
update (A, B);
update (B, {f = fun x -> x});
...
```

It clearly appears why the call to `A.f 0` raises an exception : `A` is updated before `B`, so the copied block contains `fun x -> raise (Undefined_recursive_module)` .

Compared to in-place update, relaxed in-place update allows to compile more programs and to retain the efficiency of in-place update. However, if one of the recursively-defined values is accessed before it has been updated, then an exception is raised. This maintains the soundness of the language, since it avoids segmentation faults, but weakens it, because more programs will behave badly.

2.5 Immediate in-place update

The goal of this work is to set up a type system for checking that recursive definitions are well-founded. So, a first remark is that we can get rid of a limitation of the relaxed in-place update scheme, namely the fact that only sound modules may be forward referenced. Indeed, if we succeed, we won't need to initialize the pre-allocated modules with dummy values, since the type system ensures that their fields won't be accessed before being updated with correct values. So, the only condition we should impose on forward referenced modules is to have a predictable size, in order, at least, to be able to pre-allocate a memory block for them. Fortunately, for the case of modules, the size may always be guessed from the types. Thus, all modules may be forward referenced, as long as their fields are not accessed before being updated.

Here, we consider a slightly more general setting, possibly in order to apply our type system also to the base language (thus allowing function application as right-hand sides of plain recursive definitions): we assume that not all expressions have a statically predictable size. Thus, the compilation of a list of mutually recursive definitions proceeds as follows. First, we assume that the guessable sizes have been provided by a prior static analysis, over which we completely abstract. So, definitions of predictable size are annotated with a natural number representing it. At this point, we reject programs where definitions of unpredictable size are forward referenced. If the program is correct, here is what the compiled code does. For each definition $X =_{[n]} m$ of known size n, an uninitialized memory block of size n is allocated, and bound to X in the following. Then, the first definition is computed, which gives a value v. If this definition was of known size n, then v should be a block of size n, and its pre-allocated block gets updated with v. This process is carried on for each definition. We call this method the *immediate in-place update* scheme.

An example of execution is presented in Fig. 4. The definition is $x_1 = e_1, x_2 = e_2, x_3 = e_3$, where e_1 and e_3 are expected to evaluate to blocks of sizes 2 and 1, respectively, but where the representation for the value of e_2 is not statically predictable. The pre-allocation step only allocates dummy blocks for x_1 and x_3. The value v_1 of e_1 is then computed. It can reference x_1 and x_3, which correspond to pointers to the dummy blocks, but not x_2, which would not make any sense here. This value is copied to the corresponding dummy block. Then, the value v_2 of e_2 is computed. The computation can refer to both dummy blocks, and can also strongly depend on x_1, but not on x_2. Finally, the value v_3 of e_3 is computed and copied to the corresponding dummy block.

Example 1
The program of Fig. 3, compiled by immediate in-place update, may become correct. Indeed, if `A` *is assumed to have an unpredictable size, the generated code is:*

```
(* phase 1 : pre-allocation *)
let B = alloc 1 in
(* phase 2 : evaluation and update *)
let A = B in
update (B, {f = fun x -> x});
...
```

Here, `alloc 1` *denotes an instruction that allocates a new heap block of size 1. Because* `A` *is assumed to have an unpredictable size, the evaluation of* `A` *merely creates an alias of* `B`*, which does not copy the uninitialized block. Thus, the update works for both* `A` *and* `B`*. Note however that in the context of recursive modules,* `A` *would have a known size, and the example would be rejected by our type system.*

The immediate in-place update scheme implements more definitions than previous ones. Moreover, it is also simpler that the relaxed in-place update scheme, which imposes that forward references point to modules containing only fields of functional or lazy types. This justifies the use of immediate in-place update in this paper. Namely, we formalize our type system for a language called $\lambda_\circ$, initially proposed in Hirschowitz's PhD thesis [9], which features powerful enough recursive definitions to represent OCaml's recursive modules, and which is compilable via immediate in-place update.

3 The language $\lambda_\circ$

3.1 Syntax

The syntax of $\lambda_\circ$ is defined in Fig. 5. The meta-variables X and x range over names and variables, respectively. Variables are used as binders, as usual. Names are used for accessing record fields, as an external interface to other parts of the expression. Figure 5 also recapitulates the meta-variables and notations we introduce in the remainder of this section. The syntax includes the λ-calculus constructs; variables x, abstraction $\lambda x.e$, and application $e_1\ e_2$. The language also includes records,

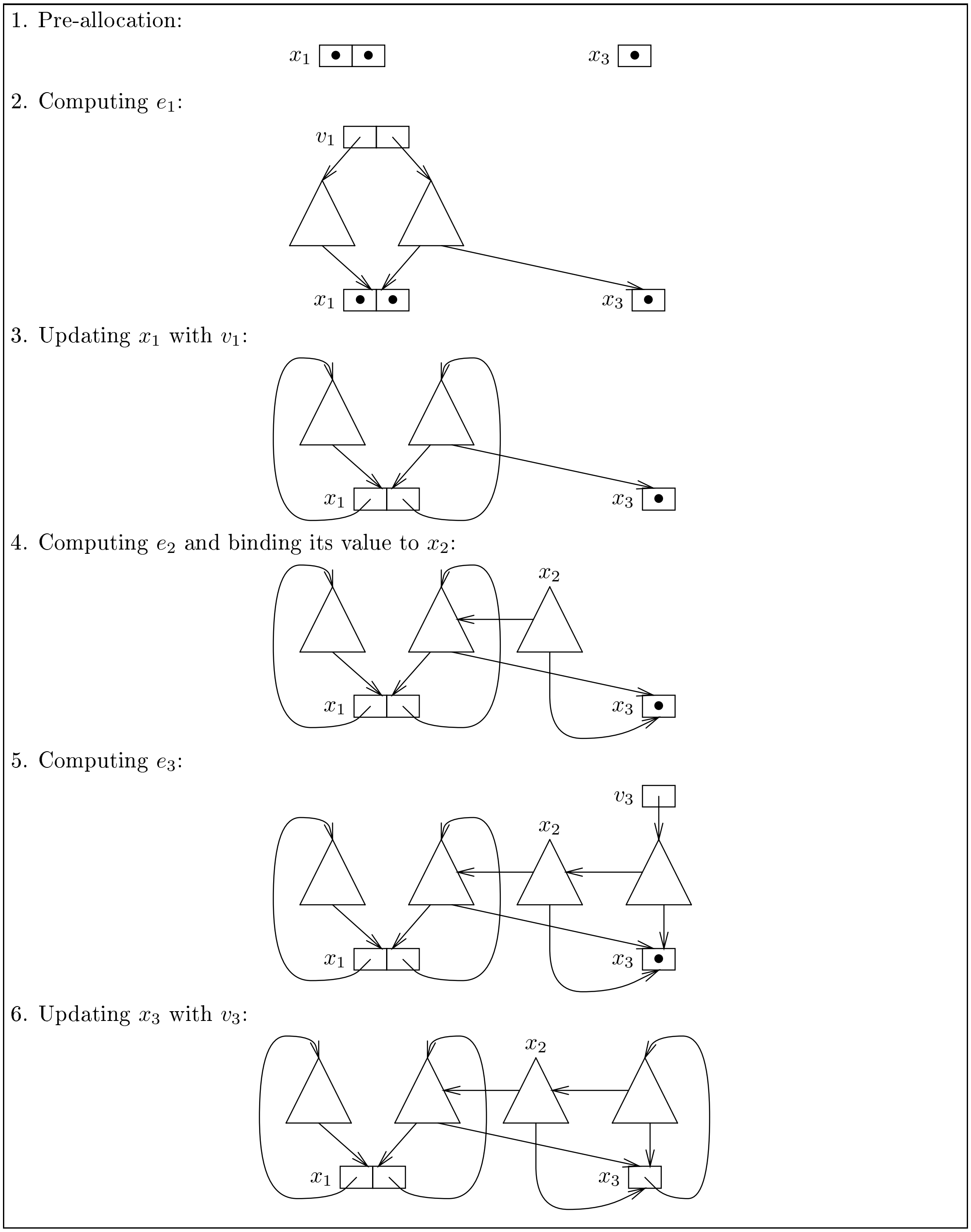


Figure 4: The refined in-place update scheme

$$
\begin{array}{lcll}
x & \in & \mathsf{Vars} & \text{Variable} \\
X & \in & \mathsf{Names} & \text{Name} \\[1ex]
\multicolumn{4}{l}{\text{Expression:}} \\
e \in \mathsf{expr} & ::= & x \mid \lambda x.e \mid e_1\ e_2 & \lambda\text{-calculus} \\
 & \mid & \{X_1 = x_1 \ldots X_n = x_n\} \mid e.X & \text{Records} \\
 & \mid & \mathtt{let\ rec}\ x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n\ \mathtt{in}\ e & \text{Recursive defs} \\[1ex]
\multicolumn{4}{l}{\text{Size indication:}} \\
\diamond & := & =_{[n]} \mid =_{[?]} & (n \text{ a natural}) \\[1ex]
\multicolumn{4}{l}{\text{Meta variables:}} \\
s & ::= & X_1 = x_1 \ldots X_n = x_n & \text{Record} \\
b & ::= & x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n & \text{Binding} \\
B & ::= & X_1 \triangleright x_1 \diamond_1 e_1 \ldots X_n \triangleright x_n \diamond_n e_n & \text{Structure} \\[1ex]
\multicolumn{4}{l}{\text{Syntactic sugar:}} \\
\multicolumn{3}{l}{\quad \mathtt{struct}\ X_1 \triangleright x_1 \diamond_1 e_1 \ldots X_n \triangleright x_n \diamond_n e_n\ \mathtt{end}} & \text{Module} \\
 & = & \mathtt{let\ rec}\ x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n & \\
 & & \mathtt{in}\ \{X_1 = x_1 \ldots X_n = x_n\} &
\end{array}
$$

Figure 5: Syntax of $\lambda_\circ$

record selection $e.X$ and a binding construct written `let rec`. To simplify the formalization and without loss of expressiveness, records are restricted to contain only variables, i.e., be of the shape $\{X_1 = x_1 \ldots X_n = x_n\}$. Bindings have the shape $\mathtt{let\ rec}\ x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n\ \mathtt{in}\ e$, where arbitrary expressions are syntactically allowed as the right-hand side of a definition, and every definition is annotated with a *size indication* $\diamond$. A size indication can be either the unknown size indication $=_{[?]}$, or a known size indication $=_{[n]}$, consisting of a natural number.

Implicit syntactic constraints Records $s = (X_1 = x_1 \ldots X_n = x_n)$ and bindings $b = (x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n)$ are required to be finite maps: a record is a finite map from names to variables, and a binding is a finite map from variables to expressions. Requiring records (resp. bindings) to be finite maps means that they should not bind the same name (resp. variable) twice. These conditions are implicitly assumed in the sequel. We refer to records and bindings collectively as *sequences*.

In a `let rec` binding $b = (x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n)$, we say that there is a *forward reference* of x_i to x_j if $i \leq j$ and $x_j \in \mathsf{FV}(e_i)$. A forward reference of x_i to x_j is syntactically forbidden, except when $\diamond_j$ is a known size indication, i.e. $\diamond_j \neq =_{[?]}$. This condition should be clear in light of the compilation scheme explained in Sect. 2. Moreover, we require that definitions of known size are not variables, that is, for each $x_i =_{[n]} e_i$ in b, e_i is not a variable.

Structural equivalence We consider expressions equivalent up to α-conversion of binding variables in functions and `let rec` expressions. In the following, except if stated otherwise, when we open a binding construct, we implicitly choose a representative witth fresh variables w.r.t. the context.

Size We have seen that `let rec`-bound definitions can be annotated with natural numbers representing their sizes. The role of these size indications is to declare in advance the expected sizes of the memory blocks representing the values of definitions. For definitions that are not forward referenced from previous definitions, there is no need for annotations. During the evaluation of a list of definitions, when the currently evaluated definition becomes a value, then its real size is matched against the syntactic size indication. If they are the same, then evaluation continues, otherwise, it gets stuck. This is why there is a notion of size in $\lambda_\circ$.

Values and answers			
$v \in \mathsf{values}$	$::=$	$x \mid \lambda x.e \mid \{s\}$	*Value*
$a \in \mathsf{answers}$	$::=$	$v \mid \texttt{let rec}\ b_v\ \texttt{in}\ v$	*Answer*

Figure 6: Answers in $\lambda_\circ$

Hypothesis 1 (Size in $\lambda_\circ$)
We assume given a partial function $\mathsf{Size}_\circ$ *from* $\lambda_\circ$ *values to natural numbers, undefined on variables.*

The hypothesis that variables have unknown sizes is related to the fact that definitions such as $x = x$ are not handled by in-place update. Our semantics therefore distinguishes variable and non-variable values.

3.2 Dynamic semantics

Values, answers, and sizes We now define the dynamic semantics of $\lambda_\circ$. As defined in Fig. 6, $\lambda_\circ$ values comprise variables, functions $\lambda x.e$ and records $\{s\}$. In $\lambda_\circ$, an evaluated definition not matching its size indication is an error, in the sense that it prevents further reductions. This behavior is enforced by not considering such definitions as valid evaluation answers: a binding defining only values is considered valid only if it *respects sizes*, in the following sense.

Definition 2 (Binding respecting sizes)
A binding b defining only values respects sizes iff for each definition $(x =_{[n]} v) \in b$, $\mathsf{Size}_\circ(v)$ *is defined and equal to n.*

We let b_v range over bindings respecting sizes, written *r.s. bindings* for conciseness. The requirement that evaluated bindings respect sizes has two immediate consequences. First, the size indications are correct for the already evaluated definitions. Second, for a definition $(x =_{[n]} e)$, the topmost constructor of the value of e must be determined by previous definitions. For instance, if $n = \mathsf{Size}_\circ(\lambda x'.x')$, then the binding $(x =_{[n]} \lambda x'.x', y =_{[n]} x)$ is valid, because the topmost constructor of the definition of y, λ, is determined by the previous definition x. On the contrary, the binding $(y =_{[n]} x, x =_{[n]} \lambda x'.x')$ is invalid: x cannot be replaced with its value, according to the reduction relation defined below. These constraints make $\lambda_\circ$ compilable with the in-place update compilation scheme.

As defined below, there is no rule for eliminating `let rec` in $\lambda_\circ$. Evaluated bindings thus remain at top-level in the expression and also in answers. They serve as a kind of heap, or recursive run-time environment. In Fig. 6, a valid answer a for the evaluation of a $\lambda_\circ$ expression is defined to be a value, possibly surrounded by an evaluated binding respecting sizes. It thus can have the shape $\texttt{let rec}\ x_1 \diamond_1 v_1 \ldots x_n \diamond_n v_n\ \texttt{in}\ v$.

Value binding Besides the non-standard notion of size, the dynamic semantics of $\lambda_\circ$ is unusual in its handling of `let rec` bindings, which is adapted from the equational theory of [1]. This theory relies on the following five fundamental equations, which resemble the rules proposed by [21].

1. The first equation is `let rec` *lifting.* It lifts a `let rec` node up one level in an expression. For example, an expression of the shape $e_1\ (\texttt{let rec}\ b\ \texttt{in}\ e_2)$ is equated with $\texttt{let rec}\ b\ \texttt{in}\,(e_1\ e_2)$.

2. The second equation is *internal merging.* In a binding, when one of the definition begins with another binding, then this binding can be merged with the enclosing one. An expression of the shape $\texttt{let rec}\ b_1, x = (\texttt{let rec}\ b_2\ \texttt{in}\ e), b_3\ \texttt{in}\ f$ is equated with $\texttt{let rec}\ b_1, b_2, x = e, b_3\ \texttt{in}\ f$, provided no variable capture occurs.

3. The third equation is *external merging*, which merges two consecutive bindings. An expression of the shape $\texttt{let rec}\ b_1\ \texttt{in}\ \texttt{let rec}\ b_2\ \texttt{in}\ e$ is equated with $\texttt{let rec}\ b_1, b_2\ \texttt{in}\ e$, provided no variable capture occurs.

Lift context:
$\mathbb{L} \quad ::= \quad \Box\ e \mid v\ \Box \mid \Box.X$
Nested lift context:
$\mathbb{F} \quad ::= \quad \Box \mid \mathbb{L}\,[\mathbb{F}\,]$
Evaluation context:
$\mathbb{E} \quad ::= \quad \mathbb{F}$
$\quad\mid\quad \texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}$
$\quad\mid\quad \texttt{let rec}\ \mathbb{B}_\diamond\,[\mathbb{F}\,]\ \texttt{in}\ e$

Binding contexts:
$\mathbb{B}_\diamond \quad ::= \quad b_v, x \diamond \Box, b$
Atomic dereferencing contexts:
$\mathbb{A} \quad ::= \quad \Box\ v \mid \Box.X$
Dereferencing contexts:
$\mathbb{D} \quad ::= \quad \mathbb{E}\,[\mathbb{A}\,]$
$\quad\mid\quad \texttt{let rec}\ \mathbb{B}_{=[n]}\ \texttt{in}\ e$

Figure 7: Evaluation contexts of $\lambda_\circ$

4. The fourth equation, *external substitution*, allows to replace variables defined in an enclosing binding with their definitions. Given a context $\mathbb{C}$, an expression of the shape $\texttt{let rec}\ b\ \texttt{in}\ \mathbb{C}\,[x]$ is equated with $\texttt{let rec}\, b\, \texttt{in}\, \mathbb{C}\,[e]$, if $x = e$ appears in b and $\mathbb{C}$ neither captures x nor the free variables of e.

5. The last equation, *internal substitution*, allows to replace variables defined in the same binding with their definitions. Given a context $\mathbb{C}$, an expression of the shape $\texttt{let rec}\ b_1, y = \mathbb{C}\,[x], b_2\ \texttt{in}\ e_1$ is equated with $\texttt{let rec}\, b_1, y = \mathbb{C}\,[e_2], b_2\, \texttt{in}\, e_1$ if $x = e_2$ appears in $b_1, y = \mathbb{C}\,[x], b_2$, and if x is not captured by $\mathbb{C}$, and no variable capture occurs.

The issue is how to arrange these operations to make the evaluation deterministic and to ensure that it reaches the answer when it exists. Our choice can be summarized as follows. First, bindings that are not at top-level in the expression must be lifted before their evaluation can begin. Thus, only the top-level binding can be evaluated. As soon as one of its definitions gets evaluated, evaluation can proceed with the next one, or with the enclosed expression if there is no definition left. If evaluation meets a binding inside the considered expression, then this binding is lifted to the top level of the expression, or just before the top-level binding if there is one. In this case, it is merged with the latter, internally or externally, according to the context. External and internal substitutions only allow to copy one of the already evaluated definitions of the top-level binding, when they are needed by the evaluation, and from left to right only.

Remark 2 (Policy on substitution and call-by-value)
The fact that substitution is only performed when needed by the evaluation does not contradict the fact that $\lambda_\circ$ is call-by-value. Indeed, only values are copied, and any expression reached by the evaluation is immediately evaluated. The fact that evaluated definitions are not immediately substituted with their values in the rest of the expression is rather a matter of presentation. In particular, it allows $\lambda_\circ$ to properly represent recursive data structures.

Our strategy is implemented by two relations: the *contraction relation* $\leadsto$, handling reductions inside the expressions, and the *reduction relation* $\longrightarrow$, handling top-level reductions. We write $\leadsto^+$ (resp. $\leadsto^*$) for the transitive (resp. reflexive transitive) closure if the relation $\leadsto$, and similarly for $\longrightarrow$.

The contraction relation The semantics of record selection and of function application are defined in Fig. 8, by *contraction* rules, defining the local *contraction relation* $\leadsto$. Record projection selects the appropriate field in the record (rule PROJECT). The application of a function $\lambda x.e$ to a value v reduces to the body of the function, where the argument has been bound to x by `let rec` (rule BETA). Rule LIFT describes how `let rec` bindings are lifted up to the top of the term. *Lift contexts* $\mathbb{L}$ are defined by

$$\mathbb{L} \quad ::= \quad \Box.X \mid \Box\ e \mid v\ \Box$$

Rule LIFT states that an expression of the shape $\mathbb{L}\,[\texttt{let rec}\ b\ \texttt{in}\ e]$ evaluates to $\texttt{let rec}\ b\ \texttt{in}\ \mathbb{L}\,[e]$, provided no variable capture occurs. For convenience, we introduce the predicate $\#$, which holds iff its two arguments are disjoints sets.

Contraction rules

$$\{s\}.X \rightsquigarrow s(X) \quad (\textsc{Project}) \qquad \frac{x \notin \mathsf{FV}(v)}{(\lambda x.e)\ v \rightsquigarrow \texttt{let rec}\ x =_{[?]} v\ \texttt{in}\ e} \quad (\textsc{Beta})$$

$$\frac{\mathsf{dom}(b) \# \mathsf{FV}(\mathbb{L})}{\mathbb{L}[\texttt{let rec}\ b\ \texttt{in}\ e] \rightsquigarrow \texttt{let rec}\ b\ \texttt{in}\ \mathbb{L}[e]} \quad (\textsc{Lift})$$

Reduction rules

$$\frac{e \rightsquigarrow e'}{\mathbb{E}[e] \longrightarrow \mathbb{E}[e']} \quad (\textsc{Context}) \qquad \frac{\mathsf{dom}(b_1) \# (\{x\} \cup \mathsf{dom}(b_v, b_2) \cup \mathsf{FV}(b_v, b_2) \cup \mathsf{FV}(e'))}{\begin{array}{c}(\texttt{let rec}\ b_v, x \diamond (\texttt{let rec}\ b_1\ \texttt{in}\ e), b_2\ \texttt{in}\ e') \\ \longrightarrow (\texttt{let rec}\ b_v, b_1, x \diamond e, b_2\ \texttt{in}\ e')\end{array}} \quad (\text{IM})$$

$$\frac{\mathsf{dom}(b) \# (\mathsf{dom}(b_v) \cup \mathsf{FV}(b_v))}{(\texttt{let rec}\ b_v\ \texttt{in}\ \texttt{let rec}\ b\ \texttt{in}\ e) \longrightarrow \texttt{let rec}\ b_v, b\ \texttt{in}\ e} \quad (\text{EM}) \qquad \frac{\mathbb{D}(x) = v}{\mathbb{D}[x] \longrightarrow \mathbb{D}[v]} \quad (\textsc{Subst})$$

Access in evaluation contexts

$$(\texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F})(x) = b_v(x) \quad (\text{EA}) \qquad (\texttt{let rec}\ b_v, y \diamond \mathbb{F}, b\ \texttt{in}\ e)(x) = b_v(x) \quad (\text{IA})$$

Figure 8: Dynamic semantics of $\lambda_\circ$

The reduction relation The reduction relation is defined in Fig. 8. Rule Context extends the contraction relation to any evaluation context. Evaluation contexts are defined in Fig. 7. We call a *nested lift context* $\mathbb{F}$ a series of lift contexts. We call a *binding context* $\mathbb{B}_\diamond$ of size $\diamond$ a binding $(b_v, x \diamond \square, b)$ where the context hole $\square$ corresponds to the next definition to be evaluated, and this definition is annotated by $\diamond$. An *evaluation context* $\mathbb{E}$ is a nested lift context, possibly appearing as the next definition to evaluate in the top-level binding, or enclosed inside a fully evaluated top-level binding. This unusual formulation of evaluation contexts enforces the determinism of the reduction relation. The idea is that evaluation never takes place inside or under a `let rec`, except the top-level one. Other bindings inside the expression first have to be lifted to the top by rule Lift, and then merged with the top-level binding if any, by rules EM and IM. If the top-level binding is of the shape $b_v, x \diamond (\texttt{let rec}\ b_1\ \texttt{in}\ e), b_2$, rule IM allows to merge b_1 with it, obtaining $b_v, b_1, x \diamond e, b_2$. When an inner binding has been lifted to the top level, if there is already a top-level binding, then the two bindings are merged together by rule EM. This implements the strategy informally described above.

Finally, rules Subst, IA, and EA describe how the variables defined by the top-level binding are replaced with their values when needed, i.e. when they appear in a *dereferencing context*. Dereferencing contexts may take two forms. They can be binding contexts of known size $\texttt{let rec}\ b_v, x =_{[n]} \square, b\ \texttt{in}\ e$. This is consistent with the fact that a definition of the shape $(x =_{[n]} y)$ is not considered fully evaluated, although y is indeed a value. Instead, the right-hand side of a definition of size n must eventually be replaced with a non-variable value of size n. Alternatively, evaluation contexts can be *atomic dereferencing contexts* wrapped by an evaluation context, i.e., contexts of the shape $\mathbb{E}[\mathbb{A}]$, where $\mathbb{A} ::= \square\ v \mid \square.X$.

In $\lambda_\circ$, the value of a variable is copied only when needed for function application or record selection. The value of a variable x is found in the current evaluation context, by looking for the first binding of x above the calling site, as formalized by the notion of access in evaluation contexts. There are two kinds of accesses.

- In the case of a context of the shape $\texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}$, if the called variable x is bound in the top-level binding b_v, then $b_v(x)$ is the requested value.

- In the case of a context of the shape $\mathbb{E}[\texttt{let rec}\ b_v, y \diamond \mathbb{F}, b\ \texttt{in}\ e]$, if the called variable x is bound in the binding b_v, then $b_v(x)$ is the requested value.

3.3 Recursive modules

Let us show simple examples showing how to encode recursive modules in $\lambda_\circ$. Other examples of $\lambda_\circ$ programs may be found elsewhere [12]. First, of course $\lambda_\circ$ allows to model mutually recursive data structures, which is necessary in order to account for recursive modules. For instance, using the syntactic sugar $\texttt{struct}\ B\ \texttt{end}$ in Fig. 5, we define two modules containing two mutually recursive functions:

$$\begin{array}{ll}
\texttt{let rec} & \mathit{Even} =_{[?]} \texttt{struct} \\
& \quad \mathit{even} \triangleright \mathit{even} =_{[?]} \lambda x.(x = 0)\ \texttt{or}\ (\mathit{Odd}.\mathit{odd}\ (x - 1)) \\
& \texttt{end}, \\
& \mathit{Odd} =_{[n]} \texttt{struct} \\
& \quad \mathit{odd} \triangleright \mathit{odd} =_{[?]} \lambda x.(x > 0)\ \texttt{and}\ (\mathit{Even}.\mathit{even}\ (x - 1)) \\
& \texttt{end} \\
\texttt{in}\ \mathit{Even}.\mathit{even}\ 56 &
\end{array}$$

(where n is assumed to be the right size indication). Notice that the function definitions and the first module do not need to have known sizes, since the only forward reference concerns the second module Odd.

This example already allows to encode some examples with recursive modules, but not all. Indeed, many practical uses of recursive modules include functor applications. For instance, consider again the example in Fig. 1. Abstracting over the static part, we encode it in $\lambda_\circ$ by

$$\begin{array}{ll}
\texttt{let rec} & A =_{[?]} \texttt{struct} \\
& \quad \mathit{compare} \triangleright \mathit{compare} =_{[?]} \ldots \mathit{ASet}.\mathit{compare} \ldots \\
& \texttt{end}, \\
& \mathit{ASet} =_{[n]} \mathit{Set}.\mathit{Make}\ A \\
\texttt{in}\ \ldots &
\end{array}$$

(where n is assumed to be the right size indication). Just like in the encoding of mixin modules, this expression evaluates correctly because $\mathit{Set}.\mathit{Make}$ only needs a pointer to its argument to return a correct result.

4 Abstract degree theory

Section 5 below defines a type system for $\lambda_\circ$, which is very expressive w.r.t. mutual dependencies, since it allows to type the bindings generated by the local fixed-point encoding of mixin modules [9]. This type system relies on a notion of degree, which is also used by the simpler type system presented in Sect. 6. Thus, we start by an abstract presentation of some theoretical properties of degrees.

4.1 Intuitions

The purpose of the degree theory introduced in the next few sections is the dependency analysis of bindings. This theory should be able, given a binding $b = (x_1 = e_1 \ldots x_n = e_n)$, to answer questions like "does the evaluation of x_i require the value of x_j?", taking into account the possibility of indirect dependency.

For instance, consider

$$\begin{array}{rl}
b = (x & = \{X = z\} \\
y & = x.X.Z \\
z & = \{Z = \{\}\}).
\end{array}$$

Directly, y strongly depends on x, and x weakly depends on z (since $\{X = z\}$ is a value). This defines the direct dependency graph of b, which does not contain any backward strong dependencies. However, the evaluation of the binding goes wrong, since it reduces in two steps to

$$
\begin{aligned}
b = (x &= \{X = z\} \\
y &= z.Z \\
z &= \{Z = \{\}\}).
\end{aligned}
$$

where y strongly depends on z. Our analysis takes such cases into account, by considering the transitive closure of the direct dependency graph. A path of the direct relation corresponds to an edge labeled with its last degree. Thus, a backward path ending with a strong dependency should be considered dangerous. We will introduce a notion of binding correctness corresponding to the absence of such paths. For mixin modules, since bindings are unordered, correctness rather corresponds to the existence of an order, such that the obtained binding is correct.

Furthermore, when b is integrated into an expression, as in $\texttt{let rec}\ b\ \texttt{in}\ e$, we must be able to consider the result as one definition of a new binding $b' = (b_1, x = (\texttt{let rec}\ b\ \texttt{in}\ e), b_2)$. The evaluation of b' makes it intuitively equivalent to $b'' = (b_1, b, x = e, b_2)$. Thus, our method to approximate the graph of b' consists in considering that of b'', and then internalize $\mathsf{dom}(b)$ into x. We do this by considering the set of paths leading from any variable y to variables $\mathsf{dom}(b) \cup \{x\}$. Roughly, for each y, we consider the path with minimum degree in this set, and represent it in the graph of b' by an edge from y to x, with this degree. This way, it takes the worst case into account, and ensures correctness.

Finally, when substitutions occur in bindings, due to the substitution rules, the dependency graph evolves. We identify an ordering on graphs, such that dependency graphs decrease along the substitution rules, and decreasing preserves correctness.

4.2 Basic definitions

We first define the notion of degree structure.

Definition 3 (Degrees)
A set Degrees *has a* structure of degrees *iff it is a lower semi-lattice (i.e., a partial order with a meet operation), and its elements are partitioned into positive and negative elements, positive ones being greater than negative ones, according to the partial order.*

We fix an arbitrary, abstract structure of degrees Degrees for this section, whose elements are denoted by χ, ordering is denoted by $\geq$, greatest lower bound operation is denoted by $\wedge$. We denote by Positive and Negative the sets of positive and negative degrees, respectively. By definition, for all $\chi_1 \in \mathsf{Positive}$ and $\chi_2 \in \mathsf{Negative}$, we have $\chi_1 \geq \chi_2$. The meta variables $\chi^{\oplus}$ and $\chi^{\ominus}$ respectively range over positive and negative degrees.

Definition 4 (Dependency graph)
A dependency graph *over a set of nodes* Nodes *is a finite subset of* $\mathsf{Nodes} \times \mathsf{Nodes} \times \mathsf{Degrees}$*, that is, a finite, oriented graph, labeled with degrees. Graphs are considered equivalent modulo the following equation:*

$$
\begin{tikzcd}
N_1 \arrow[r, bend left, "\chi_1"] \arrow[r, bend right, "\chi_2"'] & N_2
\end{tikzcd}
\quad \dashleftarrow\!\!\dashrightarrow \quad
N_1 \xrightarrow{\chi_1 \wedge \chi_2} N_2
$$

The nodes of dependency graphs are not relevant to the properties we want to establish, so we do not constrain them at all. We denote them by N, and denote finite sets of them by P. For more readability, we often write the edges of a graph $N_1 \xrightarrow{\chi} N_2$ instead of (N_1, N_1, χ), where $N_1, N_2 \in \mathsf{Nodes}$. We denote the set of nodes of a graph $\rightarrow$ by $\mathsf{Nodes}(\rightarrow)$. The set of targets of the edges of a graph $\rightarrow$ is denoted by $\mathsf{Targets}(\rightarrow)$, and similarly, sources are denoted by $\mathsf{Sources}(\rightarrow)$. The meta variable G will also range over graphs, in contexts where the graphical notation $\rightarrow$ is ill-suited.

Correctness We now define two notions of correctness for dependency graphs.

Definition 5 (Transitive closure)
We define the transitive closure on dependency graphs as the fixed-point of the operation that adds an edge $N_1 \xrightarrow{\chi_2} N_3$ for each pair of edges $N_1 \xrightarrow{\chi_1} N_2$ and $N_2 \xrightarrow{\chi_2} N_3$ in its argument $\longrightarrow$.

This fixed-point is always well-defined, since the considered operation does not introduce any degree or node, so the number of edges of the generated graphs is bounded. The transitive closure of a graph $\longrightarrow$ is written $\longrightarrow^+$.

Some notions on *paths* are defined as follows.

Definition 6 (Paths)
A path of the dependency graph $\longrightarrow$ is a possibly empty list of consecutive edges. Its length is its number of edges. If the path is not empty, then its degree is defined as the degree of its last edge. A cycle is a non-empty path whose source and target nodes are the same. We define $\longrightarrow^*$ as the set of paths of $\longrightarrow$.

We denote paths by δ, and the empty path by ϵ. The degree χ of a non-empty path δ is written as an annotation δ^χ. The concatenation of two consecutive paths is written $\delta_1; \delta_2$. For a dependency graph $\longrightarrow$, a path is also an edge of $\longrightarrow^*$. We write $N_1 \xrightarrow{\chi}^+ N_2$ for a non-empty path of degree χ from N_1 to N_2. Also, the concatenation of a non-empty path $N_1 \xrightarrow{\chi_1}^+ N_2$ and a possibly empty path δ from N_2 to N_3 is written $N_1 \xrightarrow{\chi_1}^+ N_2; \delta^{\chi_2}$, where χ_2 is χ_1 if δ is empty, and the degree of δ otherwise. Finally, when the two ends of concatenated paths or edges are syntactically the same, we merge them. For instance, the concatenation $N_1 \xrightarrow{\chi_1}^+ N_2; N_2 \xrightarrow{\chi_2}^* N_3$ is also written $N_1 \xrightarrow{\chi_1}^+ N_2 \xrightarrow{\chi_2}^* N_3$.

Let us introduce the notion of *correctness* for dependency graphs. It relies on the notion of a *weak* cycle: a cycle is weak if all its edges are labeled with positive degrees. Otherwise, the cycle is said to be *strong*.

Definition 7 (Correctness)
A dependency graph $\longrightarrow$ is correct, written $\vdash \longrightarrow$, if it does not contain any strong cycle.

This notion is related to the following notion of ordered correctness, which relies on an order over nodes. Orders on nodes are denoted by the symbol $\trianglerighteq$. Their strict versions are denoted by $\triangleright$. For any dependency graph $\longrightarrow$, let $\xrightarrow{\ominus}$ be the set of edges of $\longrightarrow$ that are labeled with negative edges. It can be seen as a binary relation on nodes. Moreover, we write $\xrightarrow{\ominus}^+$ for the relation $(\longrightarrow^+)^\ominus$ (the transitive closure has higher precedence than the degree annotation).

Definition 8 (Ordered correctness)
A dependency graph $\longrightarrow$ is correct with respect to the order $\trianglerighteq$, *or* respects the order $\trianglerighteq$, *if $\xrightarrow{\ominus}^+ \subseteq \triangleright$. We write $\vdash (\longrightarrow, \trianglerighteq)$ or $\vdash (\longrightarrow, \triangleright)$.*

We have the following equivalence.

Proposition 1 (Existence of a correct ordering)
$\vdash \longrightarrow$ iff there exists an ordering $\trianglerighteq$ on $\mathsf{Nodes}(\longrightarrow)$ such that $\vdash (\longrightarrow, \trianglerighteq)$.

To prove it, we introduce the notion of a *backward* edge and a backward path.

Definition 9 (Backward edges and paths)
Given a dependency graph $\longrightarrow$ and an order $\trianglerighteq$ on nodes, an edge $N_1 \xrightarrow{\chi} N_2$, or a path $N_1 \xrightarrow{\chi}^* N_2$ is said to be backward if $N_2 \trianglerighteq N_1$.

Thus, a graph is correct with respect to $\trianglerighteq$ if it has no backward path labeled negatively.
Preuve

- If $\vdash (\rightarrow, \trianglerighteq)$, then $\vdash\rightarrow$. By contrapositive. Assume $\rightarrow$ has a cycle with an edge of degree $\chi \in \mathsf{Negative}$. Let N be the target of this edge. Then, the transitive closure $\rightarrow^+$ of $\rightarrow$ has an edge $N \xrightarrow{\chi}^+ N$ which is backward, so $\xrightarrow{\ominus}^+$ is not included in $\triangleright$, and therefore $\vdash (\rightarrow, \trianglerighteq)$ does not hold.

- If $\vdash\rightarrow$, then any topological sort of $\rightarrow$ gives an order such that the only backward paths are in cycles, but as $\rightarrow$ is assumed correct, these paths all have positive degrees, so $\xrightarrow{\ominus}^+$ is included in $\triangleright$.

□

Subgraphs We now define some convenient notations for referring to subgraphs.

Definition 10 (Graph restriction and co-restriction)
For any dependency graph G and set of nodes P,

- *$G_{\|P}$ denotes the restriction of G to edges leading to nodes in P,*
- *${}_{P\|}G$ denotes the restriction of G to edges starting from nodes in P,*
- *$G_{\|-P}$ denotes the restriction of G to edges leading to nodes not in P,*
- *${}_{P-\|}G$ denotes the restriction of G to edges starting from nodes not in P.*
- *If $\mathcal{D}$ is a set of degrees, then $G^{\mathcal{D}}$ is the set of edges of G with labels in $\mathcal{D}$.*

As shown by the following property, these operations commute, so we write them without precedence, e.g., ${}_{P\|}G_{\|Q}$.

Proposition 2
The following equalities hold

- ${}_{P-\|}G = {}_{(\mathsf{Nodes}\setminus P)\|}G$,
- $G_{\|-P} = G_{\|(\mathsf{Nodes}\setminus P)}$,
- $({}_{P\|}G)_{\|Q} = {}_{P\|}(G_{\|Q})$,
- ${}_{P\|}(G^{\mathcal{D}}) = ({}_{P\|}G)^{\mathcal{D}}$,
- $(G^{\mathcal{D}})_{\|P} = (G_{\|P})^{\mathcal{D}}$.

Definition 11 (Concatenation of graphs)
The concatenation $G_1; G_2$ of two dependency graphs is the set of edges $\{N_1 \xrightarrow{\xi_2} N_3 \mid N_1 \xrightarrow{\xi_1}_{G_1} N_2 \xrightarrow{\xi_2}_{G_2} N_3\}$.

Graph splitting Next, we define an operation called splitting on dependency graphs, that redirects the edges leading to a node toward another node. This notion is technically useful in the soundness proof.

Definition 12 (Graph splitting)
Let $G_{N_1 \succ N_2} = (G_{\|-\{N_1\}}) \cup \{(N_3, N_2, \chi) \mid (N_3, N_1, \chi) \in G\}$.

Internalizable graph We also need the notion of *internalizable graph*, which is used for handling the dependencies of bindings.

Definition 13 (Internalizable graph)
Let P be a set of nodes and a node $N \notin P$. A dependency graph G is termed internalizable *at P with entry point N if ${}_{P\|}G \subseteq G_{\|P\cup\{N\}}$.*

Graphs without degrees

Definition 14 (Unlabeled graphs as dependency graphs)
An unlabeled graph is viewed as a dependency graph by considering all its edges labeled with a unique, negative degree.

Unlabeled graphs have an interesting property w.r.t. ordered correctness.

Proposition 3 (Ordered correctness of unlabeled graphs)
For any order on nodes $\trianglerighteq$ *and unlabeled graph* $G = G_1 \cup G_2$, $\vdash (G, \trianglerighteq)$ *iff* $\vdash G_1 \trianglerighteq$ *and* $\vdash G_2 \trianglerighteq$.

Preuve A backward path can only be backward if there is at least one backward edge. $\square$

4.3 Graph comparison

Definition 15 (Graph comparison)
We define $\rightarrow_1 \sqsubseteq \rightarrow_2$ *as: for all* $N_1 \xrightarrow{\chi}_2 N_2$, *there exists* $\chi' \leq \chi$ *such that* $N_1 \xrightarrow{\chi'}{}^{+}_{1} N_2$.

Definition 16 (Graph strong comparison)
We define $\rightarrow_1 \sqsubseteq^* \rightarrow_2$ *by*

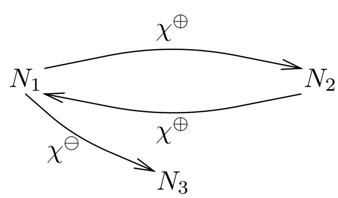
- *for all* $N_1 \xrightarrow{\chi^\ominus}_2 N_2$, *there exists* $\chi \leq \chi^\ominus$ *such that* $N_1 \xrightarrow{\chi}{}^{+}_{1} N_2$
- *and for all* $N_1 \xrightarrow{\chi^\oplus}_2 N_2$, *there exists* $\chi \leq \chi^\oplus$ *such that* $N_1 \xrightarrow{\chi}_1 N_2$.

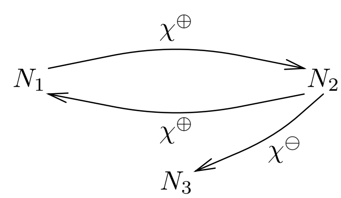
We say that $\rightarrow_1$ *is* more restrictive *than* $\rightarrow_2$.

Proposition 4

- *If* $\rightarrow_1 \sqsubseteq^* \rightarrow_2$, *then* $\rightarrow_1 \sqsubseteq \rightarrow_2$.
- *If* $\rightarrow_2 \subseteq \rightarrow_1$, *then* $\rightarrow_1 \sqsubseteq^* \rightarrow_2$.
- *If for all* $N_1 \xrightarrow{\chi}_2 N_2$ *there exists* $\chi' \leq \chi$ *such that* $N_1 \xrightarrow{\chi'}_1 N_2$, *then* $\rightarrow_1 \sqsubseteq^* \rightarrow_2$.

Notice that these relations are transitive and reflexive, but not antisymmetric, as shown by the following two graphs, which are related by $\sqsubseteq^*$ and $\sqsupseteq^*$, but are obviously not the same.





We have the intuitive property that a less restrictive graph respects all the orders a more restrictive one respects.

Proposition 5
If $\rightarrow_1 \sqsubseteq \rightarrow_2$, *and* $\vdash (\rightarrow_1, \trianglerighteq)$, *then* $\vdash (\rightarrow_2, \trianglerighteq)$.

Preuve When $\rightarrow_1 \sqsubseteq \rightarrow_2$, for each path in the transitive closure of $\rightarrow_2$, there is a path with a smaller degree in the transitive closure of $\rightarrow_1$. $\square$

Type:	$\tau ::= \tau_1 \xrightarrow{\xi} \tau_2$	Annotated function type
	$\mid \{X_1 : \tau_1 \ldots X_n : \tau_n\}$	Record type
Generalized degree:	$\xi \in \mathsf{GDegrees} ::= \text{-}\infty \mid n \mid \infty$	

Figure 9: Syntax of $\lambda_\circ$ types

Minimum (commutative)

$$\begin{array}{lcrcl} \xi & \wedge & \infty & = & \xi \\ \xi & \wedge & \text{-}\infty & = & \text{-}\infty \\ m & \wedge & n & = & \min_{\mathbb{N}}(m, n) \end{array}$$

Composition

$$\begin{array}{lcrcll} \xi & @ & \text{-}\infty & = & \text{-}\infty & \\ \xi & @ & \infty & = & \infty & \\ \xi & @ & \xi' & = & \xi & \text{if } \xi' \neq \infty, \text{-}\infty \end{array}$$

Substitution

$$\begin{array}{rcll} \xi\{\infty\} & = & \infty & \\ \xi\{\xi'\} & = & \xi & \text{if } \xi' \neq \infty \end{array}$$

Lambda

$$\begin{array}{rcrcl} \infty & \oplus & n & = & \infty \\ \text{-}\infty & \oplus & n & = & n \\ m & \oplus & n & = & m +_{\mathbb{N}} n \end{array}$$

Step

$$\begin{array}{lcll} \mathsf{if}_{\text{-}\infty}(\text{-}\infty) & = & \text{-}\infty & \\ \mathsf{if}_{\text{-}\infty}(\xi) & = & \infty & \text{if } \xi \neq \text{-}\infty \end{array}$$

Application

$$\begin{array}{rcrcll} \infty & \ominus & n & = & \infty & \\ \text{-}\infty & \ominus & n & = & \text{-}\infty & \\ m & \ominus & n & = & m -_{\mathbb{N}} n & \text{if } m > n \\ m & \ominus & n & = & \text{-}\infty & \text{if } m \leq n \end{array}$$

Plus

$$\begin{array}{lcccl} \text{-}\infty & + & 1 & = & \text{-}\infty \\ \infty & + & 1 & = & \infty \\ n' & + & 1 & = & n' +_{\mathbb{N}} 1 \end{array}$$

Minus

$$\begin{array}{rcccl} 0 & - & 1 & = & \text{-}\infty \\ \text{-}\infty & - & 1 & = & \text{-}\infty \\ \infty & - & 1 & = & \infty \\ (n +_{\mathbb{N}} 1) & - & 1 & = & n \end{array}$$

Figure 10: Operations on generalized degrees

5 A powerful type system

We now equip $\lambda_\circ$ with a sound type system that guarantees that all recursive definitions are correct, and that they match the expected sizes. Boudol [2] goes toward such a type system, however his proposal does not handle sizes, resulting in a less efficient compilation scheme [3], and it does not type-check curried function applications with sufficient precision for our purposes. Indeed, curried function applications like $(\lambda x.\lambda y.\lambda z.x\ y\ z)\ x\ y$ are considered to strongly depend on x, which prevents expressions generated by the local fixed-point encoding to be well-typed. Hirschowitz and Leroy [10] define a refined type system handling curried function applications, but not handling sizes either. Hence, we now define a further refinement of these type systems, that allows both powerful recursive definitions and sizes.

Types Types, written τ, have the syntax defined in Figure 9. Arrow types are annotated with *generalized degrees* ξ, indicating how a function uses its argument. (The name of "generalized degrees" is in relation with Boudol's notion of degree, which are generalized by this notion.) For instance, a function such as $\lambda x.x + 1$ has type $\mathtt{int} \xrightarrow{\text{-}\infty} \mathtt{int}$, because the value of x is immediately needed after application, whereas $\lambda xyz.x + 1$ has type $\mathtt{int} \xrightarrow{2} \ldots$, because the value of x is not needed unless at least 2 more function applications are performed. We define an order on generalized degrees, and show that they have a degree structure, with -∞ as unique negative degree. (We call generalized degrees simply degrees in the sequel, the distinction with the degrees of MM should be clear from the context.)

Definition 17 (Ordering generalized degrees)
Define the order on generalized degrees as the smallest reflexive, transitive relation such that for any $\xi \in \mathsf{GDegrees}$ *and* $n \in \mathbb{N}$*:*

$$\text{-}\infty \leq \xi \leq \infty$$
$$n \leq n+1.$$

We denote by $\xi_1 \wedge \xi_2$ the greatest lower bound of two generalized degrees ξ_1 and ξ_2.

Proposition 6 (Degree structure)
Generalized degrees have a structure of degree with $\mathsf{Negative} = \{\text{-}\infty\}$.

The typing judgment is of the form $\Gamma \vdash e : \tau \ / \ \gamma$, where Γ is an *environment*, that is, a finite map from variables to types, and γ is a (total) mapping from variables to degrees, called a *degree environment*. It indicates how e uses each variable: intuitively,

- $\gamma(x) = 0$ means that $e = x$, or that $e = \{\ldots X = x \ldots\}$ (x is used only as a pointer);
- $\gamma(x) = \infty$ means that x is not free in e;
- $\gamma(x) = \text{-}\infty$ means that e strongly depends on x;
- and $\gamma(x) = n+1$ means that the value of x is needed only after $n+1$ function applications, e.g., x occurs in e under at least $n+1$ function abstractions.

The restriction $\gamma_{|P}$ of a degree environment γ to a set of variables P is the function that returns $\gamma(x)$ on any $x \in P$, and ∞ on any $x \notin P$. The co-restriction $\gamma_{\backslash P}$ is defined conversely. The *support* $\mathsf{supp}(\gamma)$ of a degree environment γ is the set $\mathsf{Vars} \setminus \gamma^{-1}(\infty)$ of variables of degree different from ∞. We impose that degree environments be of finite support: for all degree environment γ, the set $\mathsf{supp}(\gamma) = \{x \in \mathsf{Vars} \mid \gamma(x) \neq \infty\}$ is finite. The *range* $\mathsf{rng}(\gamma)$ of a degree environment is defined as usual.

Finally, we make two additional hypotheses related to types. First, we assume that functions and records value have known sizes.

Hypothesis 2 (Size of functions and records $\lambda_\circ$)
We assume that for any s, $\mathsf{Size}_\circ(\{s\}) \neq =_{[?]}$.

Second, we assume that the size of values can be guessed from their types.

Hypothesis 3 (Size of types)
We assume given a total function $\mathsf{TSize}_\circ$ *from* $\lambda_\circ$ *types to size indications. By abuse of notation, known size indications are identified with the natural number they carry.*

The blocks corresponding to values of some given type τ must all have size $\mathsf{TSize}_\circ(\tau)$, when it is known. This will be enforced below by Hypothesis 4.

Typing rules The type system for $\lambda_\circ$ is defined in Figure 11, using some notions defined by cases in Figure 10.

Rule T-Var expresses that the variable x is not protected by any function abstraction via the side condition $\gamma(x) \leq 0$.

Function abstraction (rule T-Abs) increments by 1 the degree of all variables appearing in its body, except for its formal parameter x, whose degree is retained in the type of the function. We write $\gamma \ominus 1$ for the function $y \mapsto \gamma(y) \ominus 1$, where degree subtraction is defined in Figure 10. Notice that $1 \ominus 1 = \text{-}\infty$, which can be surprising. In fact, it simply states that after one application, a variable protected by one function abstraction is not considered protected anymore, as appears in $\lambda x.x + 1$ for instance.

Rule T-App deals with function application. In the function part e_1, all variable degrees are decremented by 1, since the application removes one level of abstraction. The degrees of the argument part e_2 are combined with the ξ annotation on the arrow type of e_1 via the @ operation, defined in Figure 10. Intuitively, it represent the contribution of the free variables of the argument to their degrees in the application. Because of call-by-value, strong dependencies in e_2 ($\gamma_2(x) = \text{-}\infty$)

$$\frac{\gamma(x) \leq 0}{\Gamma \vdash x : \Gamma(x) \;/\; \gamma} \;(\text{T-Var}) \qquad \frac{\Gamma + \{x : \tau'\} \vdash e : \tau \;/\; (\gamma \ominus 1)\langle x \mapsto \xi \rangle}{\Gamma \vdash \lambda x.e : \tau' \xrightarrow{\xi} \tau \;/\; \gamma} \;(\text{T-Abs})$$

$$\frac{\Gamma \vdash e_1 : \tau' \xrightarrow{\xi} \tau \;/\; \gamma_1 \qquad \Gamma \vdash e_2 : \tau' \;/\; \gamma_2 \qquad \gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2}{\Gamma \vdash e_1 \; e_2 : \tau \;/\; \gamma} \;(\text{T-App})$$

$$\frac{\mathsf{dom}(I) = \mathsf{dom}(s) \qquad \gamma' \leq (\gamma + 1) \qquad \forall X \in \mathsf{dom}(s), \Gamma \vdash s(X) : I(X) \;/\; \gamma}{\Gamma \vdash \{s\} : \{I\} \;/\; \gamma'} \;(\text{T-Record})$$

$$\frac{X \in \mathsf{dom}(I) \qquad \gamma \leq \gamma' - 1 \qquad \Gamma \vdash e : \{I\} \;/\; \gamma'}{\Gamma \vdash e.X : I(X) \;/\; \gamma} \;(\text{T-Select})$$

$$\frac{\begin{array}{c} \mathit{res} \text{ a fresh variable} \qquad \Gamma + \Gamma_b \vdash e : \tau \;/\; \gamma_e \\ \Gamma + \Gamma_b \vdash b : \Gamma_b \;/\; G \qquad \vdash_{\lambda_\circ} (G, b) \qquad \gamma \leq (G \cup (\gamma_e \longrightarrow \mathit{res}) \gg \mathsf{dom}(b) \gg \mathit{res}) \end{array}}{\Gamma \vdash \texttt{let rec } b \texttt{ in } e : \tau \;/\; \gamma} \;(\text{T-LetRec})$$

$$\Gamma \vdash \epsilon : \epsilon \;/\; \emptyset \;(\text{T-Empty}) \qquad \frac{\Gamma \vdash e : \Gamma(x) \;/\; \gamma \qquad \Gamma \vdash b : \Gamma_b \;/\; G}{\Gamma \vdash (x =_{[?]} e, b) : \Gamma_b + \{x : \Gamma(x)\} \;/\; G \cup (\gamma \longrightarrow x)} \;(\text{T-Unknown})$$

$$\frac{\Gamma \vdash e : \Gamma(x) \;/\; \gamma \qquad \Gamma \vdash b : \Gamma_b \;/\; G \qquad \gamma^{-1}(0) = \emptyset \qquad \mathsf{TSize}_\circ(\Gamma(x)) = =_{[n]}}{\Gamma \vdash (x =_{[n]} e, b) : \Gamma_b + \{x : \Gamma(x)\} \;/\; G \cup (\gamma \longrightarrow x)} \;(\text{T-Known})$$

Figure 11: Typing rules for $\lambda_\circ$

remain strong in the application: $\xi \mathbin{@} \text{-}\infty = \text{-}\infty$ for any ξ. Variables not free in e_2 ($\gamma_2(x) = \infty$) do not contribute any dependency to the application. The interesting case is that of a variable x with degree $\xi' \neq \infty, \text{-}\infty$ in e_2, i.e. not immediately needed. We do not know how many times the function e_1 is going to apply its argument inside its body. However, we know that it will not do so before ξ more applications of $e_1 \; e_2$. Hence, we can take ξ for the degree of x in $e_1 \; e_2$. Finally, the contributions from the function part ($\gamma_1 \ominus 1$) and the argument part ($\xi \mathbin{@} \gamma_2$) are combined with the $\wedge$ operator, which is point-wise minimum.

Remark 3 (Another explanation)
The $\xi_1 \mathbin{@} \xi_2$ operation could be defined as $\mathsf{if}_{\text{-}\infty}(\xi_2) \wedge \xi_1\{\xi_2\}$, where substitution $\cdot\{\cdot\}$ and step $\mathsf{if}_{\text{-}\infty}()$ are defined in Figure 10. The degree environment in the conclusion of the T-App *can also be written $(\gamma_1 \ominus 1) \wedge \mathsf{if}_{\text{-}\infty}(\gamma_2) \wedge \xi\{\gamma_2\}$.*

Explanation:

- *the function part loses one level of abstraction, whence the $\gamma_1 \ominus 1$ part;*
- *the argument to the function must be computed, and this is represented by the $\mathsf{if}_{\text{-}\infty}(\gamma_2)$ part;*
- *then the argument to the function replaces a variable of degree ξ, as indicated by the type of the function. The operation $\cdot\{\cdot\}$ computes an approximation of the resulting degree environment. When a variable has no free occurrence in the argument, one can safely give it the degree ∞. When it has an occurrence in the argument, we reason as follows.*
  - *If $\xi = \text{-}\infty$, then the body of the function uses the argument, and we do not know how many times it could apply it, so it is possibly more than the degree of any variable in γ_2.*
  - *If $\xi = n + 1$, roughly, we know that we can safely apply the result n times, but then, it works exactly as above, we do not know how the argument is going to be used, so we are limited to approximating all the degrees in $\mathsf{supp}(\gamma_2)$ at $n + 1$.*

- *If $\xi = 0$, then we approximate all the degrees in* $\mathsf{supp}(\gamma_2)$ *at* 0.

If $\xi = 0$, in fact, we could sharpen our approximation: in principle, it indicates that the body of the function is more or less the argument variable, so we should be able to reuse the degrees of the argument exactly. This would give the rule $0 \mathbin{@} \xi = \xi$. There are at least two reasons for not doing this.

- *First, we will see that a rather standard weakening property on degrees holds: one can replace degrees with inferior degrees in a typing judgment, without breaking its derivability. The proposed rule however, has strange consequences: it is no longer true that the @ operation is monotone in both of its arguments. Indeed, we have $0 \leq 3$, but $3 = 3 \mathbin{@} 5 \leq 0 \mathbin{@} 5 = 5$. The consequences of this are uncertain.*
- *Second, and it is certainly related to the first reason, sharpening our approximation makes it too sharp for dependency graphs. Throughout the thesis, we have fixed that the notion of transitive closure of dependency graphs only takes into account the degree of the last edges of paths. Consider a simple* Beta *contraction step: $e_1 = (\lambda x.x)(\lambda y.z)$ is contracted to $e_2 = \texttt{let rec}\ x =_{[?]} \lambda y.z\ \texttt{in}\ x$. With the proposed rule, the degree of z in e_1 is $\infty \wedge 0 \mathbin{@} 1 = 1$. However, in e_2, we will see below that one has to consider the following dependency graph, where res is a fresh variable:*

$$z \xrightarrow{\quad 1 \quad} x \xrightarrow{\quad 0 \quad} res$$

It has a path from z to res, of degree 0, so the degree of z in e_2 is 0 at most. Thus, the proposed rule breaks type preservation, unless we change the notion of transitive closure for dependency graphs. This is unnecessary for our purposes.

The rule for record construction (rule T-Record) is straightforward, since it does not modify the degree environment. The rule for selection (rule T-Select) types the components of the record with a common degree environment γ', which must not give degree 0 to any variable. The final degree environment has to be inferior to γ'. The rationale for the restriction of γ' is explained by the following example. Consider $e = x.X$. If we omit the restriction, e has type τ in any environment Γ such that $\Gamma(x) = \tau$, in the degree environment $\gamma = \{x \mapsto 0\}$, whereas the degree of x in e ought to be -∞. For instance, consider the expression $e' = (\lambda y.\{\})x.X$. By rule T-App, it is well-typed in the empty degree environment $\{z \mapsto \infty \mid z \in \mathsf{Vars}\}$, but is stuck since no value can be found for x.

The most complex rule is T-LetRec for mutually recursive definitions. For typing a `let rec` expression $\texttt{let rec}\ b\ \texttt{in}\ e$, the T-LetRec rule introduces a typing environment Γ_b with domain $\mathsf{dom}(b)$, and adds it to the initial environment Γ. In this enriched environment, it is checked that e has the final type τ, yielding a degree environment γ_e. Then, the typing of b is delegated to the dedicated judgment, consisting of rules T-Empty, T-Unknown, and T-Known. For each definition $(x \diamond_x e)$ of the binding, these rules ensure that e has the expected type, yielding degree environments γ_x. While typing the binding, the rules build a dependency graph G, equal to $\bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x)$, where $\gamma \longrightarrow x$ denotes $\{y \xrightarrow{\gamma(y)} x \mid \gamma(y) \neq \infty\}$. Simultaneously, the rules check that for each definition of known size $x =_{[n]} e$, $\Gamma_b(x)$ is indeed of size n, and also ${\gamma_x}^{-1}(0) = \emptyset$. This last verification is not very intuitive at first sight, but can be understood as follows. In $\lambda_\circ$, bindings must respect sizes, in the sense of Section 3.2. Thus, for a binding like $(b, x =_{[n]} y, b')$, y should not be defined in b'. Indeed, it would make the intended value of x in the prefix $(b, x =_{[n]} y)$ undefined. The condition ${\gamma_x}^{-1}(0) = \emptyset$ ensures that this cannot happen. Indeed, when typing the definition $x =_{[n]} y$, y could either have degree -∞ or 0, but with the condition, it only can have type -∞, which forces y to be defined before x.

Once the binding is typed, rule T-LetRec checks that the graph G is compatible with the order of definition and the size indications in b, which is denoted by $\vdash_{\lambda_\circ} (G, b)$. Let $\rhd_b$ denote the order of definitions in b, and $\vdash_{\lambda_\circ} (G, b)$ mean that $\vdash (G, \rhd_b)$, in the sense of ordered correctness (see Definitions 8 and 14).

An important remark is that the absence of backward dependencies on definitions of unknown size is trivially preserved by reduction, because dependencies become less backward along the reduction. Thus, we do not need to check it explicitly in the type system, since it is part of syntactic correctness.

Finally, the whole expression is given type τ, in a complex degree environment, computed from the graph $G \cup (\gamma_e \longrightarrow res)$. The expression `let rec` b `in` e can depend on a variable y in several ways.

- The variable y can have an occurrence in e directly. Then, edges of $\gamma_e \longrightarrow res$ from y to res model this dependency.

- Also, it can have an occurrence in one of the bindings of b, say x. Then, paths of the graph starting with this edge $y \longrightarrow x$ model this dependency. Paths leading to res must be reflected in the final degree environment, but paths stopping at a binding in b only need to be reflected if they have a negative degree. Indeed, they cannot be part of any cycle.

Formally, the degree environment for the whole expression is required to be no greater than to $(G \gg \mathsf{dom}(b) \gg res)$, where $(G \gg \mathsf{dom}(b) \gg res)$ is the degree environment *internalizing* $\mathsf{dom}(b)$ into res, defined as follows, for any general degree structure, on any set of nodes.

Definition 18 (Internalized degree environment)
Let G be a graph internalizable at P with entry point N. The degree environment internalizing P into N in G is

$$\begin{aligned}(G \gg P \gg N) = \quad & \bigwedge_{N_2 \in P} \{N_1 \mapsto \chi^{\ominus} \mid (N_1 \xrightarrow{\chi^{\ominus}} N_2) \in {}_{P-\|}((G_{\|P})^{+})\} \\ & \wedge \{N_1 \mapsto \chi \mid (N_1 \xrightarrow{\chi} N) \in {}_{P-\|}((G_{\|P})^{*}; G_{\|\{N\}})\}.\end{aligned}$$

The graph ${}_{P-\|}((G_{\|P})^{+})$ is the set of edges of $(G_{\|P})^{+}$, i.e. the set of non-empty paths of $G_{\|P}$, that do not begin with a node in P. In other terms, each edge of ${}_{P-\|}((G_{\|P})^{+})$ corresponds to an edge of ${}_{P-\|}G_{\|P}; (G_{\|P})^{*}$ or equivalently ${}_{P-\|}G_{\|P}; ({}_{P\|}G_{\|P})^{*}$. In fact, as ${}_{P\|}G \subseteq G_{\|P \cup \{N\}}$, it can also be viewed as ${}_{P-\|}G_{\|P}; (G_{\|-\{N\}})^{*}$.

As announced, we assume that the function giving the size of types returns the right sizes.

Hypothesis 4 (Size of types)
We assume that the function $\mathsf{TSize}_{\circ}$, from $\lambda_{\circ}$ types to size indications, is such that if $\Gamma \vdash v : \tau$, and v is not a variable, then $\mathsf{TSize}_{\circ}(\tau) = \mathsf{Size}_{\circ}(v)$.

We now state some useful elementary lemmas.

The standard type weakening and strengthening lemmas are straightforward.

Lemma 1 (Type environment weakening)
If $\Gamma \vdash e : \tau \;/\; \gamma$, and $\mathsf{dom}(\Gamma') \mathrel{\#} \mathsf{FV}(e)$, then $\Gamma + \Gamma' \vdash e : \tau \;/\; \gamma$.

Lemma 2 (Type environment strengthening)
If $\Gamma + \Gamma' \vdash e : \tau \;/\; \gamma$, and $\mathsf{dom}(\Gamma') \mathrel{\#} \mathsf{FV}(e)$, then $\Gamma \vdash e : \tau \;/\; \gamma$.

We then remark that the typing judgment still holds if the degree environment γ is replaced by another environment $\gamma' \leq \gamma$, or if the degree $\gamma(x)$ of an unused variable x is changed.

Lemma 3 (Degree environment weakening)
If $\gamma' \leq \gamma$ and $\Gamma \vdash e : \tau \;/\; \gamma$, then $\Gamma \vdash e : \tau \;/\; \gamma'$.

Now, we prove that the only necessary information provided by degree environments concerns the free variables of the considered expression. Other informations could be ignored.

Lemma 4 (Degree environment strengthening)
If $\Gamma \vdash e : \tau \;/\; \gamma$, $P \supseteq \mathsf{FV}(e)$ and $\gamma'_{\mid P} = \gamma_{\mid P}$, then $\Gamma \vdash e : \tau \;/\; \gamma'$.

Finally, we state two lemmas that are useful for proving the soundness of the simpler type system presented in the next section.

$$
\begin{array}{lll}
\text{Type:} & \sigma ::= \sigma_1 \to \sigma_2 & \text{Function type} \\
 & \mid \{X_1 : \sigma_1 \ldots X_n : \sigma_n\} & \text{Record type} \\
 & \mid [\sigma_1 \ldots \sigma_n] \Rightarrow \sigma' & \text{Safe function type}
\end{array}
$$

Figure 12: Syntax of simplified $\lambda_\circ$ types

Lemma 5 **(n abstractions)**
The following typing rule is admissible for the type system of $\lambda_\circ$.

$$
\frac{\Gamma + \{x_1 : \tau_1 \ldots x_n : \tau_n\} \vdash e : \tau \;/\; (\gamma \ominus n)\langle x_1 \mapsto \xi_1 \ldots x_n \mapsto \xi_n\rangle}{\Gamma \vdash \vec{\lambda}(x_1, \ldots, x_n).e : \tau_1 \xrightarrow{\xi_1 \oplus (n-1)} \tau_2 \xrightarrow{\xi_2 \oplus (n-2)} \ldots \tau_n \xrightarrow{\xi_n} \tau \;/\; \gamma}
$$

Lemma 6 **(n applications)**
The following typing rule is admissible for the type system of $\lambda_\circ$, provided for all i, $\xi_i \neq 0$.

$$
\frac{\Gamma \vdash e : \tau_1 \xrightarrow{\xi_1 \oplus (n-1)} \tau_2 \xrightarrow{\xi_2 \oplus (n-2)} \ldots \tau_n \xrightarrow{\xi_n} \tau \;/\; \gamma \qquad \Gamma(x_i) = \tau_i \textit{ for } i = 1, \ldots, n}{\Gamma \vdash e(x_1, \ldots, x_n) : \tau \;/\; (\gamma \ominus n) \wedge \{x_1 \mapsto \xi_1 \ldots x_n \mapsto \xi_n\}}
$$

6 A simpler type system

The previous type system is too complex to be implemented as such, for instance in the OCaml module system. We propose a more pratical restriction, which handles the most common cases.

This system relies on depedency graphs with a less expressive set of degrees: strong depedencies are quoted by $\bullet$, and weak ones by $\circ$, which give us a new set of degrees with only two elements and with $\circ \geq \bullet$.

6.1 Type system

Types Types have the syntax defined in Fig.12. A function has a type $[\tau_1 \ldots \tau_n] \Rightarrow \tau'$ when the values of its n arguments are not needed after n function applications. Otherwise it has type $\tau_1 \to \tau_2$. For instance:

- $\lambda x.x + 1$ has type $\texttt{int} \to \texttt{int}$
- $\lambda xy.\texttt{let rec } z = x + 1 \texttt{ in } z + y$ has type $\texttt{int} \Rightarrow \texttt{int} \to \texttt{int}$: the value of x is not needed after one application, but the value of y is needed after 2 applications
- $\lambda xy.\texttt{let rec } z = x + 1 \texttt{ in } \lambda t.t + z + y$ has type $\texttt{int} \Rightarrow \texttt{int} \Rightarrow \texttt{int} \to \texttt{int}$: the value of x is not needed after one application but needed after 2 applications, the value of y is not needed after 2 applications
- $\lambda xy.\texttt{let rec } f = \lambda z.x + 1 \texttt{ in } \lambda t.t + y + x$ has type $[\texttt{int}, \texttt{int}] \Rightarrow \texttt{int} \to \texttt{int}$: the values of x and y are not needed after 2 function applications

Typing rules The typing rules are given in Fig. 13.To check if a function is "safe", we consider the depedency graph of its body: if there is no strong depedency in one of its variables (i.e. if there is no path ended with a $\bullet$ with its source in $\{x_1 \ldots x_n\}$), the function is typed $[\sigma_1 \ldots \sigma_n] \Rightarrow \tau$ (rule Safe-Abs). The binding b of a $\texttt{let rec}$ b $\texttt{in}$ e is also checked (rule LetRec). As a convenient notation, we write $\tau_1 \ldots \tau_n$ for the corresponding list of types, empty if $n = 0$. Similarly, we write $[\tau_1 \ldots \tau_n] \Rightarrow \tau$ for τ if $n = 0$, and the expected type otherwise. In rule App, the notation $\sigma' \Rrightarrow \sigma$ denotes either $\sigma' \to \sigma$, or $[\sigma', \sigma_1 \ldots \sigma_n] \Rightarrow \sigma_0$, with $\sigma = [\sigma_1 \ldots \sigma_n] \Rightarrow \sigma_0$. Similarly, we denote some combination of safe and unsafe function types by $[\sigma_1 \ldots \sigma_n] \Rrightarrow \sigma_0$. Also, we write $\Gamma \vdash e : \sigma :: \gamma$ for $\Gamma \vdash e : \sigma$ and $\Gamma \vdash e :: \gamma$.

Expressions $\boxed{\Gamma \vdash e : \sigma}$

$$\Gamma \vdash x : \Gamma(x) \quad (\textsc{Var}) \qquad \frac{\mathsf{rng}(s) \subseteq \Gamma}{\Gamma \vdash \{s\} : \Gamma \circ s} \quad (\textsc{Record}) \qquad \frac{\Gamma \vdash e : \{I\}}{\Gamma \vdash e.X : I(X)} \quad (\textsc{Select})$$

$$\frac{\Gamma + \{x : \sigma'\} \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \sigma' \to \sigma} \quad (\textsc{Unsafe-Abs}) \qquad \frac{\Gamma \vdash e_1 : \sigma' \Rrightarrow \sigma \qquad \Gamma \vdash e_2 : \sigma'}{\Gamma \vdash e_1\ e_2 : \sigma} \quad (\textsc{App})$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma + \{x_1 : \sigma_1 \ldots x_n : \sigma_n\} \qquad n \geq 1 \\ \Gamma' \vdash \texttt{struct}\ B\ \texttt{end} : \sigma \qquad \Gamma' \vdash \mathsf{Bind}(B) :: G \qquad \mathsf{Sources}(G^{+\bullet}) \subseteq \mathsf{DV}(B)\end{array}}{\Gamma \vdash \lambda x_1 \ldots x_n . \texttt{struct}\ B\ \texttt{end} : [\sigma_1 \ldots \sigma_n] \Rightarrow \sigma} \quad (\textsc{Safe-Abs})$$

$$\frac{\Gamma + \Gamma_b \vdash e : \sigma \qquad \Gamma + \Gamma_b \vdash b : \Gamma_b \qquad \Gamma \vdash b :: G \qquad G^{+\bullet}{}_{\parallel \mathsf{dom}(b)} \subseteq \rhd_b}{\Gamma \vdash \texttt{let rec}\ b\ \texttt{in}\ e : \sigma} \quad (\textsc{LetRec})$$

Bindings $\boxed{\Gamma \vdash b : \Gamma_b}$

$$\Gamma \vdash \emptyset : \epsilon \quad (\textsc{Empty}) \qquad \frac{\Gamma \vdash e : \Gamma(x) \qquad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x =_{[?]} e, b) : \Gamma_b + \{x : \Gamma(x)\}} \quad (\textsc{Unknown})$$

$$\frac{\Gamma \vdash e : \Gamma(x) \qquad \Gamma \vdash b : \Gamma_b \qquad \mathsf{TSize}_\circ(\Gamma(x)) = =_{[n]}}{\Gamma \vdash (x =_{[n]} e, b) : \Gamma_b + \{x : \Gamma(x)\}} \quad (\textsc{Known})$$

Expression depedency graph $\boxed{\Gamma \vdash e :: \gamma}$

$$\frac{v \text{ is not a variable}}{\Gamma \vdash v :: \circ_{\mid \mathsf{FV}(v)}} \quad (\textsc{G-Weak}) \qquad \Gamma \vdash e :: \bullet_{\mid \mathsf{FV}(e)} \quad (\textsc{G-Strong})$$

$$\frac{p \geq 0 \qquad n \geq m \geq 1 \qquad e = x.X_1 \ldots X_p \qquad \Gamma \vdash e : [\sigma_1 \ldots \sigma_n] \Rightarrow \sigma}{\Gamma \vdash (e\ x_1 \ldots x_m) :: (\bullet_{\mid \{x\}} \wedge \circ_{\mid \{x_1 \ldots x_m\}})} \quad (\textsc{G-App})$$

$$\frac{res \notin \mathsf{dom}(b) \cup \mathsf{FV}(b, e) \qquad \Gamma \vdash (b, res =_{[?]} e) :: G \qquad G^{+\bullet}{}_{\parallel \mathsf{dom}(b)} \subseteq \rhd_b}{\Gamma \vdash \texttt{let rec}\ b\ \texttt{in}\ e :: {}_{\mathsf{FV}(\texttt{let rec}\ b\ \texttt{in}\ e) \parallel} G^{+}} \quad (\textsc{G-Struct})$$

Binding depedency graph $\boxed{\Gamma \vdash b :: G}$

$$\frac{\forall x \in \mathsf{dom}(b), \Gamma \vdash b(x) :: \gamma_x}{\Gamma \vdash b :: \bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x)} \quad (\textsc{G-Bind})$$

Figure 13: Simple typing rules for $\lambda_\circ$

Types

$$
\begin{array}{lcl}
[\![\{I\}]\!] & = & \{[\![I]\!]\} \\
[\![\sigma' \to \sigma]\!] & = & [\![\sigma']\!] \xrightarrow{-\infty} [\![\sigma]\!] \\
[\![[\sigma_1 \dots \sigma_n] \Rightarrow \sigma]\!] & = & [\![\sigma_1]\!] \xrightarrow{n} \dots \xrightarrow{2} [\![\sigma_n]\!] \xrightarrow{1} [\![\sigma]\!]
\end{array}
$$

Degrees

$$
\begin{array}{lcl}
[\![\bullet]\!] & = & -\infty \\
[\![\circ]\!] & = & 1
\end{array}
$$

Degree environments

$$
\begin{array}{lcll}
[\![\gamma]\!](x) & = & [\![\gamma(x)]\!] & \text{if } x \in \mathsf{dom}(\gamma) \\
[\![\gamma]\!](x) & = & \infty & \text{if } x \notin \mathsf{dom}(\gamma)
\end{array}
$$

Graphs

$$
[\![G]\!] \;=\; \{x \xrightarrow{[\![\chi]\!]} y \mid x \xrightarrow{\chi}_G y\}
$$

Figure 14: Translation

Depedency graph construction rules Variables are generally typed $\bullet$, except if (Fig. 13):

- it is protected by an abstraction (rule G-ABS)
- it is an argument of a function typed $[\sigma_1 \dots \sigma_n] \Rightarrow \sigma$ (rule G-APP)

6.2 Soundness

In this section we will prove the soundness of this system by injection into the previous one. The translation is given in Fig. 14.

We state first a trivial proposition (the depedency graph construction rules give directly the proof).

Proposition 7 (Shape of the graph of an expression)
If $\Gamma \vdash e :: \gamma$, then $\mathsf{dom}(\gamma) = \mathsf{FV}(e)$ and $\gamma \le \circ_{\mid \mathsf{FV}(e)}$. Further, we have the following.

- *If e is of the shape $e'.X$ or is a variable, then $\gamma = \bullet_{\mid \mathsf{FV}(e)}$.*
- *Let $m > 0$ and $p \ge 0$. If e is of the shape $(x.X_1 \dots X_p\ x_1 \dots x_m)$, then $\gamma \le \bullet_{\mid \{x\}} \wedge \circ_{\mid \{x_1 \dots x_m\}}$.*

We also remark that the translation never gives degree 0 to any variable.

Proposition 8 (Degrees given by the translation)
For all depedency graph γ, $\mathsf{rng}([\![\gamma]\!]) \subseteq \{-\infty, 1, \infty\}$.

Theorem 1 (Soundness of the simple type system)
If $\Gamma \vdash e :: \gamma$ and $\Gamma \vdash e : \sigma$ then $[\![\Gamma]\!] \vdash e : [\![\sigma]\!] \mathrel{/} [\![\gamma]\!]$.
If $\Gamma \vdash b :: G$ and $\Gamma \vdash b : \Gamma_b$ then $[\![\Gamma]\!] \vdash b : [\![\Gamma_b]\!] \mathrel{/} [\![G]\!]$.

Preuve The proof is by mutual induction on e and b, and case analysis on the last typing rule.

- VAR

  We have $e = x$, then by Prop. 7 we have $[\![\gamma]\!] = -\infty_{\mid \{x\}}$, then by rule T-VAR, we have $[\![\Gamma]\!] \vdash x : [\![\Gamma]\!](x) \mathrel{/} [\![\gamma]\!]$.

- UNSAFE-ABS

  We have that e is an expression of the shape $\lambda x.e'$, $\sigma = \sigma_1 \to \sigma_2$, and $\gamma \le \circ_{\mid \mathsf{FV}(e)}$ and $\mathsf{dom}(\gamma) = \mathsf{FV}(e)$, by Prop. 7. So we have $([\![\gamma]\!] \ominus 1)\langle x \mapsto -\infty\rangle = -\infty_{\mid \mathsf{FV}(e) \cup \{x\}}$. Let $\gamma' = \bullet_{\mid \mathsf{FV}(e')}$; we have $[\![\gamma']\!] \ge ([\![\gamma]\!] \ominus 1)\langle x \mapsto -\infty\rangle$ and $\Gamma \vdash e' :: \gamma'$. By induction we have $[\![\Gamma + \{x : \sigma_1\}]\!] \vdash e' : [\![\sigma_2]\!] \mathrel{/} [\![\gamma']\!]$, which by Lemma 3 gives $[\![\Gamma + \{x : \sigma_1\}]\!] \vdash e' : [\![\sigma_2]\!] \mathrel{/} ([\![\gamma]\!] \ominus 1)\langle x \mapsto -\infty\rangle$, so by rule T-ABS, we have $[\![\Gamma]\!] \vdash \lambda x.e : [\![\sigma_1]\!] \xrightarrow{-\infty} [\![\sigma_1]\!] \mathrel{/} [\![\gamma]\!]$.

- App

  We have that e is an expression of the shape $e_1\ e_2$, and $\Gamma \vdash e_1 : \sigma' \Rrightarrow \sigma$, and $\Gamma \vdash e_2 : \sigma'$. We proceed by case analysis on the last rule of the derivation of $\Gamma \vdash e :: \gamma$. We have two cases:

  - G-App. In this case, e_1 has the shape $e_3\ x_1 \ldots x_{m-1}$, with $n \geq m \geq 1$, $e_2 = x_m$ and $e_3 = x.X_1 \ldots X_p$, and we have $\Gamma \vdash e_3 : [\sigma_1, \ldots \sigma_n] \Rightarrow \sigma''$. So, we have some record types $I_1 \ldots I_p$, such that $\Gamma(x) = \{I_1\}$, and for all $i \in \{1 \ldots p-1\}$, $I_i.X_i = \{I_{i+1}\}$, and $I_p(X_p) = [\sigma_1, \ldots \sigma_n] \Rightarrow \sigma$. But obviously, this type is the only derivable type for e_3, so $\sigma' = \sigma_m$, $\sigma' \Rrightarrow \sigma = [\sigma_m, \ldots \sigma_n] \Rightarrow \sigma''$, and $\sigma = [\sigma_{m+1}, \ldots \sigma_n] \Rightarrow \sigma''$.

    By rule T-Var, we obtain $[\![\Gamma]\!] \vdash x : \{[\![\cdot]\!] \circ I_1\} \ / \ \text{-}\infty_{|\,\{x\}}$, and then by successive application of rule T-Select, $[\![\Gamma]\!] \vdash e_3 : [\![\sigma_1]\!] \xrightarrow{n} \ldots \xrightarrow{2} [\![\sigma_n]\!] \xrightarrow{1} [\![\sigma'']\!] \ / \ \text{-}\infty_{|\,\{x\}}$. Finally, by Lemma 6, we obtain $[\![\Gamma]\!] \vdash e : [\![\sigma]\!] \ / \ \text{-}\infty_{|\,\{x\}} \wedge 1_{|\,\{x_1 \ldots x_m\}}$, which is the expected result.

  - Otherwise, the last rule applied is rule G-Strong, and we have $\gamma = \bullet_{|\,\mathsf{FV}(e)}$. By induction we have $[\![\Gamma]\!] \vdash e_1 : [\![\sigma']\!] \xrightarrow{-\infty} [\![\sigma]\!] \ / \ [\![\gamma_1]\!]$ and $[\![\Gamma]\!] \vdash e_2 : [\![\sigma]\!] \ / \ [\![\gamma_2]\!]$, with $\gamma_i = \bullet_{|\,\mathsf{FV}(e_i)}$. But $[\![\gamma]\!] = \text{-}\infty_{|\,\mathsf{FV}(e)}$, so $[\![\gamma]\!] = ([\![\gamma_1]\!] \ominus 1) \wedge \text{-}\infty \mathbin{@} [\![\gamma_2]\!]$, then by rule T-App, we have $[\![\Gamma]\!] \vdash e_1\ e_2 : [\![\sigma]\!] \ / \ [\![\gamma]\!]$.

- Record

  We have a derivation of the shape:

$$\frac{\mathsf{rng}(s) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \{s\} : \Gamma \circ s}$$

  For all $X \in \mathsf{dom}(s)$, by rule T-Var, we have $[\![\Gamma]\!] \vdash s(X) : [\![\Gamma]\!](s(X)) \ / \ \gamma'$, with $\gamma' = 0_{|\,\mathsf{rng}(s)}$. But by Prop. 7 we have $\gamma \leq \circ_{|\,\mathsf{FV}(\{s\})}$. So $[\![\gamma]\!] \leq 1_{|\,\mathsf{FV}(e)} = \gamma' + 1$, so by T-Record, we have $[\![\Gamma]\!] \vdash \{s\} : [\![\Gamma]\!] \circ s \ / \ [\![\gamma]\!]$.

- Select

  We have that e is of the shape $e'.X$, and by Prop. 7, $\gamma = \bullet_{|\,\mathsf{FV}(e)} = \bullet_{|\,\mathsf{FV}(e')}$. We have $\Gamma \vdash e' :: \gamma$ by rule G-Strong, so by induction we have $[\![\Gamma]\!] \vdash e' : \{[\![I]\!]\} \ / \ [\![\gamma]\!]$. So by T-Select, $[\![\Gamma]\!] \vdash e'.X : [\![I(X)]\!] \ / \ [\![\gamma]\!] - 1$. But we have $[\![\gamma]\!] - 1 = \text{-}\infty_{|\,\mathsf{FV}(e')} - 1 = \text{-}\infty_{|\,\mathsf{FV}(e')} = [\![\gamma]\!]$, so we have $[\![\Gamma]\!] \vdash e'.X : [\![I(X)]\!] \ / \ [\![\gamma]\!]$.

- Safe-Abs

  By Prop. 7, we have $\gamma \leq \circ_{|\,\mathsf{FV}(e)}$. By inversion, we have $e = \lambda x_1 \ldots x_n.\mathtt{struct}\ B\ \mathtt{end}$ and $\Gamma' = \Gamma + \{x_1 : \sigma_1 \ldots x_n : \sigma_n\}$, such that $\Gamma' \vdash \mathtt{struct}\ B\ \mathtt{end} : \sigma'$, with $\sigma = [\sigma_1 \ldots \sigma_n] \Rightarrow \sigma'$. Furthermore, by typing of $\mathtt{struct}\ B\ \mathtt{end}$, we have G such that $\Gamma' \vdash b :: G$ with $b = \mathsf{Bind}(B)$, and $\mathsf{Sources}(G^{+\bullet}) \subseteq \mathsf{DV}(B)$. Finally, by rule Record, we have $\Gamma' \vdash \{s\} :: \gamma_s$ with $\gamma_s = \circ_{|\,\mathsf{FV}(\{s\})}$.

  Thus, taking res a fresh variable and letting $H = G \cup (\gamma_s \longrightarrow res)$, we have by rule G-Struct that $\Gamma' \vdash \mathtt{struct}\ B\ \mathtt{end} :: {}_{\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})||}H^+$.

  But in fact, the graphs ${}_{\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})||}H^+$ and ${}_{\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})||}G^+$ are equal. Indeed, consider any path $x \xrightarrow{\chi}{}^{+}_{H} y$, with $x \in \mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})$. If this path contains no edge to res, then it is in ${}_{\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})||}G^+$. Otherwise,

  - there is no other edge to res,
  - it is the last edge of the path,
  - and $\chi = \circ$, by construction of H.

  But since $\mathsf{rng}(s) \mathbin{\#} \mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})$, this last edge is not the only edge of the path, so the path can be decomposed into $x \xrightarrow{\chi_1}{}^{+}_{G} z \xrightarrow{\chi}_{H} y$. So, there is a path with source x in G, whose degree is necessarily less than or equal to $\circ$, which gives the expected result.

Then, since $\mathsf{Sources}(G^{+\bullet}) \subseteq \mathsf{DV}(B)$, we obtain that as a degree environment, ${}_{\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})\|}G^{+} = \circ_{|\,\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})}$, so we derive $\Gamma' \vdash \mathtt{struct}\ B\ \mathtt{end} :: \circ_{|\,\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})}$, and by induction hypothesis, we have $[\![\Gamma']\!] \vdash \mathtt{struct}\ B\ \mathtt{end} : [\![\sigma']\!] \ /\ 1_{|\,\mathsf{FV}(\mathtt{struct}\ B\ \mathtt{end})}$.

So, by Lemma 5, we have exactly $[\![\Gamma']\!] \vdash e : [\![\sigma]\!] \ /\ (n+1)_{|\,\mathsf{FV}(e)}$, which gives the expected result by weakening.

- LetRec

  We have that e is of the shape $\mathtt{let\ rec}\ b\ \mathtt{in}\ e'$. By typing hypothesis, we have Γ_b and G such that letting $\Gamma' = \Gamma + \Gamma_b$, $\Gamma' \vdash b : \Gamma_b$, $\Gamma' \vdash e' : \sigma$, and $\Gamma \vdash b :: G$.

  Moreover, by the second hypothesis, we get H and a fresh variable res such that $\Gamma \vdash (b, res {=_{[?]}} e') :: H$. By construction, this H can be seen as the union of G' and $\gamma' \longrightarrow res$ such that ${G'^{+\bullet}}_{\|\,\mathsf{dom}(b)} \subseteq \rhd_b$, $\Gamma \vdash b :: G'$ and $\Gamma \vdash e' :: \gamma'$, which by weakening implies $\Gamma' \vdash b :: G'$ and $\Gamma' \vdash e' :: \gamma'$.

  By induction hypothesis, this gives $[\![\Gamma']\!] \vdash b : [\![\Gamma_b]\!] \ /\ [\![G']\!]$ and $[\![\Gamma']\!] \vdash e' : [\![\sigma]\!] \ /\ [\![\gamma']\!]$. Moreover, $[\![G'^{+\bullet}]\!] = [\![G']\!]^{+\text{-}\infty}$, so $\vdash_{\lambda_\circ} (G', b)$.

  Finally, we have $[\![\gamma]\!] = [\![{}_{\mathsf{FV}(e)\|}H^{+}]\!] = {}_{\mathsf{FV}(e)\|}[\![H]\!]^{+} \leq ([\![H]\!] \gg \mathsf{dom}(b) \gg res)$. Indeed, by definition of a degree environment internalization, given $x \in \mathsf{dom}([\![H]\!] \gg \mathsf{dom}(b) \gg res)$, we have $x \in \mathsf{FV}(e)$ and there are two possibilities: either there exists $y \in \mathsf{dom}(b)$ such that $x \xrightarrow{\text{-}\infty}{}^{+}_{[\![H]\!]} y$, or $x \xrightarrow{\xi}{}^{+}_{[\![H]\!]} res$. In both cases, we have a path with the same ends and at most the same degree in ${}_{\mathsf{FV}(e)\|}[\![H]\!]^{+}$.

  Thus, we can apply rule T-LetRec so derive $[\![\Gamma]\!] \vdash e : [\![\sigma]\!] \ /\ [\![\gamma]\!]$.

- Unknown and Known

  Easy by construction of the graphs of bindings and Prop. 7.

□

A Generalized degrees

We now turn to proving the soundness of the proposed type system.

A.1 Simple properties of generalized degrees

We start the proof with a number of algebraic lemmas on degrees and degree operations. The following lemmas should be read as universally quantified over the degrees ξ, ξ', ξ_1, ξ_2, ξ_3. We adopt the convention that @ has highest precedence, followed by $\wedge$, and then $\oplus$ and $\ominus$.

Lemma 7

1. $(\xi_1 \oplus 1) \mathbin{@} \xi_2 \leq \xi_1 \mathbin{@} \xi_2 \oplus 1$.
2. $(\xi_1 \wedge \xi_2) \mathbin{@} \xi_3 = \xi_1 \mathbin{@} \xi_3 \wedge \xi_2 \mathbin{@} \xi_3$.
3. $\xi_1 \mathbin{@} (\xi_2 \wedge \xi_3) = \xi_1 \mathbin{@} \xi_2 \wedge \xi_1 \mathbin{@} \xi_3$.
4. $(\xi_1 \mathbin{@} \xi_2) \mathbin{@} \xi_3 = \xi_1 \mathbin{@} (\xi_2 \mathbin{@} \xi_3)$.
5. $(\xi \ominus n) \mathbin{@} \xi' = \xi \mathbin{@} \xi' \ominus n$.
6. $\text{-}\infty \ominus 1 \oplus 1 = 1$, $0 \ominus 1 \oplus 1 = 1$, $0 \oplus 1 \ominus 1 = \text{-}\infty$.
7. *If* $\xi \neq 0$, *then* $\xi \oplus 1 \ominus 1 = \xi$.
8. *If* $\xi \notin \{\text{-}\infty, 0\}$, *then* $\xi \ominus 1 \oplus 1 = \xi$.

9. $-\infty \mathbin{@} \xi \leq \xi$.

10. *If* $\xi \leq \xi'$ *then* $\xi \oplus 1 \leq \xi' \oplus 1$ *and* $\xi \ominus 1 \leq \xi' \ominus 1$.

11. *If* $\xi \oplus 1 \leq \xi' \ominus 1$ *then* $\xi \oplus 2 \leq \xi'$.

12. $(\xi_1 \wedge \xi_2) \ominus 1 = (\xi_1 \ominus 1) \wedge (\xi_2 \ominus 1)$.

13. *If* $\xi_1 \leq \xi_2$, *then* $\xi \mathbin{@} \xi_1 \leq \xi \mathbin{@} \xi_2$.

14. $\xi + 1 - 1$, $\xi - 1 \leq \xi$, $(\xi \wedge \xi') - 1 = (\xi - 1) \wedge (\xi' - 1)$.

Preuve

1. If $\xi_2 = -\infty$, we obtain $-\infty \leq 1$ which is true. If $\xi_2 = \infty$ we obtain $\infty \leq \infty$. Otherwise, the claim reduces to $\xi_1 \oplus 1 \leq \xi_1 \oplus 1$.

2. If $\xi_3 = -\infty$, we obtain $-\infty$ on both sides of the equality. If $\xi_3 = \infty$, both sides are equal to ∞. Otherwise we get $\xi_1 \wedge \xi_2$ on both sides.

3. If $\xi_2 = -\infty$, both sides are equal to $-\infty$. If $\xi_2 = \infty$, then $\xi_2 \wedge \xi_3 = \xi_3$ and $\xi_1 \mathbin{@} \xi_2 = \infty$, so both sides are equal to $\xi_1 \mathbin{@} \xi_3$. Otherwise, we argue by case on ξ_3. If $\xi_3 = -\infty$, then we obtain $-\infty$ on both sides, and if $\xi_3 = \infty$, we obtain $\xi_1 \mathbin{@} \xi_2$ for both sides. Otherwise, $\xi_2 \wedge \xi_3 = n \neq -\infty$, so $\xi_1 \mathbin{@} (\xi_2 \wedge \xi_3) = \xi_1 = \xi_1 \wedge \xi_1 = \xi_1 \mathbin{@} \xi_2 \wedge \xi_1 \mathbin{@} \xi_3$.

4. If $\xi_3 = -\infty$, both sides are equal to $-\infty$. If $\xi_3 = \infty$, we obtain ∞ on both sides. Otherwise, both sides are equal to $\xi_1 \mathbin{@} \xi_2$.

5. Both sides reduce to ∞ if $\xi' = \infty$, to $-\infty$ if $\xi' = -\infty$, and to $\xi \ominus n$ otherwise.

6. By definition of $\oplus$ and $\ominus$.

7. By definition of $\oplus$ and $\ominus$.

8. By definition of $\oplus$ and $\ominus$.

9. If $\xi = -\infty$ or $\xi = \infty$, both sides of the inequality are equal to ξ. Otherwise, the inequality is equivalent to $-\infty \leq \xi$, which is true for any ξ.

10. By definition of $\oplus$ and $\ominus$.

11. Since $\xi \oplus 1 \geq 1$, $\xi' \ominus 1 \geq 1$ so $\xi' \geq 1$, and so by Item 8, $\xi' = \xi' \ominus 1 \oplus 1$, and the result follows by applying Property 10 to $\xi \oplus 1 \leq \xi' \ominus 1$.

12. By symmetry, we can assume w.l.o.g. that $\xi_1 \leq \xi_2$. Then $\xi_1 \wedge \xi_2 = \xi_1$, and by Property 10, $(\xi_1 \ominus 1) \wedge (\xi_2 \ominus 1) = \xi_1 \ominus 1$.

13. If $\xi_2 = \infty$, then $\xi \mathbin{@} \xi_2 = \infty$, which is necessarily greater than $\xi \mathbin{@} \xi_1$. Otherwise, if $\xi_2 = -\infty$, then $\xi_1 = -\infty$, and $\xi \mathbin{@} \xi_1 = \xi \mathbin{@} \xi_2 = -\infty$. Otherwise, ξ_2 is a natural number, so $\xi \mathbin{@} \xi_2 = \xi$. But as $\xi_1 \leq \xi_2$, $\xi_1 \neq \infty$. Hence, if $\xi_1 = -\infty$, then $\xi \mathbin{@} \xi_1 = -\infty$, which is necessarily inferior to $\xi \mathbin{@} \xi_2$. Otherwise, ξ_1 and ξ_2 are both natural numbers, so $\xi \mathbin{@} \xi_1 = \xi = \xi \mathbin{@} \xi_2$.

14. By definition.

□

A.2 Internal merging

We use internalized degree environments to model the IM reduction rule, at the level of dependency graphs. Given G, internalizable at P into N, internalizing P into N in G also corresponds to an operation on G. The link is made by the following definition of *internalized graph*.

Definition 19 (Internalized graph)
Let G be a dependency graph internalizable at P with entry point N. Let the graph internalizing P into N in G be

$$\mathsf{Internalize}(G, P, N) = {}_{P-\|}G_{\|-P\cup\{N\}} \cup ((G \gg P \gg N) \longrightarrow N).$$

The connection with internalized degree environments is explained below.

Remark 4 (Forgotten dependency paths)
Internalized degree environments do not take into account the dependency paths stopping before the final target N that have a positive degree. This refinement is mostly technical, and we can explain it with an example. For example, let $e = \texttt{let rec}\ x =_{[n]} (\lambda y.\lambda z.\{\})\ x\ \texttt{in}\ \{\}$. We certainly want e to be well-typed, and expect x to have degree ∞ in the expression $(\lambda y.\lambda z.\{\})\ x$. This is what the type system does, since $\infty @ 0 = \infty$. However, consider now the reduct $e' = \texttt{let rec}\ x =_{[n]} (\texttt{let rec}\ y =_{[?]} x\ \texttt{in}\ (\lambda z.\{\}))\ \texttt{in}\ \{\}$. If the operation $(\cdot \gg \cdot \gg \cdot)$ took all dependencies into account, the degree of x in $\texttt{let rec}\ y =_{[?]} x\ \texttt{in}\ (\lambda z.\{\})$ would be 0, thus breaking type preservation.

Notice that a variable of degree ∞ can occur free in the considered expression. This is harmless though, since these forgotten dependencies cannot be part of any cycle: other definitions than x cannot depend on y, and the intended one for x does not depend on y either (it is equal to $\lambda z.\{\}$). However, this phenomenon forces us to eliminate backward dependencies on definitions of unknown size independently, although one could have hoped that it could be done by examining dependency graphs.

In this section, we prove some properties of internalized graphs and degree environments. For more generality, we reason on an arbitrary set of nodes, although the considered degree structure remains that of generalized degrees. So, a degree environment is a total function from nodes to generalized degrees, of finite support. Moreover, we remark that generalized degrees respect the following hypothesis.

Proposition 9 (Unique negative degree)
We assume that the considered degree structure has a unique negative degree.

Internalized degree environments and graphs give environments and graphs that contain enough information for ensuring the correctness of the initial graphs, in the sense that they detect all dependencies cycles containing negative degrees, and that they correctly predict the dependencies after reduction. This is shown by the next two lemmas, which use the following property.

Proposition 10 (Complete internalized graph)
Let G be a graph internalizable at P with entry point N. For all path $N_1 \xrightarrow{\xi}{}^{+}_{G} N_2$ of G, with $N_1 \notin P$,

- *either $N_2 \in P$ and $\xi \in \mathsf{Positive}$,*
- *or $N_2 \in P$, $\xi \in \mathsf{Negative}$ and $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N$,*
- *or $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N_2$.*

Preuve We proceed by induction on the number of edges not in ${}_{P-\|}G_{\|-P\cup\{N\}}$.

- If all edges are in ${}_{P-\|}G_{\|-P\cup\{N\}}$, then, trivially $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N_2$.

- If $n+1$ edges are outside ${}_{P-\|}G_{\|-P\cup\{N\}}$, let $N_3 \xrightarrow{\xi_2}_G N_4$ be the first of these. We have $N_1 \xrightarrow{\xi}{}^{+}_{G} N_2 = N_1 \xrightarrow{\xi_1}{}^{*}_{G} N_3 \xrightarrow{\xi_2}_G N_4 \xrightarrow{\xi}{}^{*}_{G} N_2$, with $N_1 \xrightarrow{\xi_1}{}^{*}_{\mathsf{Internalize}(G,P,N)} N_3$. We know that $N_3 \notin P$ and $N_4 \in P \cup \{N\}$. We distinguish the following three cases.
  - $N_4 = N$. Then, $(G \gg P \gg N)(N_3) = \xi_2$, so $N_3 \xrightarrow{\xi_2}_{\mathsf{Internalize}(G,P,N)} N_4$, and we conclude by induction hypothesis on $N_4 \xrightarrow{\xi}{}^{*}_{G} N_2$.
  - The rest of the path, $N_4 \xrightarrow{\xi}{}^{*}_{G} N_2$, contains only nodes in P, i.e.it is a path of ${}_{P\|}G_{\|P}$.
    * If $\xi \in \mathsf{Negative}$, then $N_3 \xrightarrow{\xi}_{\mathsf{Internalize}(G,P,N)} N$, so $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N$: we are in the second case.
    * Otherwise, we are in the first case.
  - The rest of the path $N_4 \xrightarrow{\xi}{}^{*}_{G} N_2$ is of the shape $N_4 \xrightarrow{\xi_3}{}^{*}_{{}_{P\|}G_{\|P}} N_5 \xrightarrow{\xi_4}_{{}_{P\|}G_{\|-P}} N_6 \xrightarrow{\xi}{}^{*}_{G} N_2$. By hypothesis, as ${}_{P\|}G_{\|-P} = {}_{P\|}G \cap G_{\|-P}$, we have ${}_{P\|}G_{\|-P} \subseteq G_{\|P\cup\{N\}} \cap G_{\|-P} = G_{\|\{N\}}$, so $N_6 = N$, and $N_3 \xrightarrow{\xi_4}_{\mathsf{Internalize}(G,P,N)} N_6$. If the last path $N_6 \xrightarrow{\xi}{}^{*}_{G} N_2$ is empty, then $\xi_4 = \xi$, and we are in the third case. Otherwise, by induction hypothesis, there are three possibilities.
    * $N_6 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N_2$, and $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N_2$: we are in the third case.
    * $N_2 \in P$ and $\xi \in \mathsf{Positive}$, and we are in the first case.
    * $N_2 \in P$, $\xi \in \mathsf{Negative}$, and $N_6 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N$, and $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N$: we are in the second case.

□

Lemma 8 (Complete internalized graph)
Let G be a graph internalizable at P with entry point N. Let $\trianglerighteq$ be a total order on $\mathsf{Nodes}(G)$*, such that*

- $P \triangleright N$,
- *for all $N' \notin P$, if $N' \triangleright N$, then $N' \triangleright P$,*
- $\vdash (\mathsf{Internalize}(G,P,N), \trianglerighteq)$,
- *and* $\vdash ({}_{P\|}G_{\|P}, \trianglerighteq)$.

We have $\vdash (G, \trianglerighteq)$.

Preuve We prove that all negative paths of G are forward. Let $N_1 \xrightarrow{\xi}{}^{+}_{G} N_2$ a path of G, with $\xi \in \mathsf{Negative}$.

- If $N_1 \notin P$, by Property 10, and as $\xi \in \mathsf{Negative}$, there are two possibilities.
  - $N_2 \in P$ and $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N$. By hypothesis, $\vdash (\mathsf{Internalize}(G,P,N), \triangleright)$, so $N_1 \triangleright N$, and, by hypothesis, this implies that $N_1 \triangleright P$, so $N_1 \triangleright N_2$ and the considered path is forward.
  - $N_1 \xrightarrow{\xi}{}^{+}_{\mathsf{Internalize}(G,P,N)} N_2$, so $N_1 \triangleright N_2$ by correctness of $\mathsf{Internalize}(G,P,N)$, and the considered path is forward.
- If $N_1 \in P$.
  - If all nodes of the considered path are in P, then it is included in ${}_{P\|}G_{\|P}$, and is forward by hypothesis.
  - Otherwise, consider the first node not in P: it is necessarily N because ${}_{P\|}G \subseteq G_{\|P\cup\{N\}}$. So the considered path is of the shape $N_1 \xrightarrow{\xi_1}{}^{*}_{{}_{P\|}G_{\|P\cup\{N\}}} N \xrightarrow{\xi}{}^{*}_{G} N_2$.

* If the second part of the path, $N \xrightarrow{\xi}{}^{*}_{G} N_2$, is empty, then then considered path is from $N_1 \in P$ to N, and by hypothesis $N_1 \rhd N$.
* Otherwise, the same reasoning as above demonstrates that $N \rhd N_2$, so by transitivity of $\rhd$, we have $N_1 \rhd N_2$.

□

Lemma 9 (Complete internalized degree environment)
Let P and Q be disjoint sets of nodes, $N \in Q$, and G a dependency graph internalizable at P with entry point N and internalizable at $P \cup Q$ with entry point N_0, such that $\mathsf{Targets}(G) \subseteq P \cup Q \cup \{N_0\}$, *and* $\mathsf{Sources}(G) \# \{N_0\}$. *We have*

$$(\mathsf{Internalize}(G, P, N) \gg Q \gg N_0) \leq (G \gg (P \cup Q) \gg N_0).$$

Preuve By hypothesis, we have $N_0 \notin P \cup Q$, and ${}_{P\|}G \subseteq G_{\|P \cup \{N\}}$, ${}_{P \cup Q\|}G \subseteq G_{\|P \cup Q \cup \{N_0\}}$. Let $\gamma = (\mathsf{Internalize}(G, P, N) \gg Q \gg N_0)$ and $\gamma' = (G \gg P \cup Q \gg N_0)$. Let $G_1 = \mathsf{Internalize}(G, P, N)$. First, notice that the conditions on G imply that $\mathsf{Targets}(G_1) \subseteq Q \cup \{N_0\}$, and $\mathsf{Sources}(G_1) \# \{N_0\}$. Consequently, ${}_{Q\|}G_1 \subseteq G_{1\|Q \cup \{N_0\}}$. So the two degree environments are well-defined.

For any node N_1, let $\xi = \gamma'(N_1)$, and let $R = P \cup Q$. There are two possibilities, by definition of $(\cdot \gg \cdot \gg \cdot)$.

- $N_1 \xrightarrow{\xi_1}_{R-\|G\|R} N_2 \xrightarrow{\xi}{}^{*}_{R\|G\|R} N_3$ and $\xi \in \mathsf{Negative}$. We have $N_1 \notin P$, so by Property 10, there are two possibilities.
  - $N_3 \in P$ and $N_1 \xrightarrow{\xi}{}^{+}_{G_1} N$, so, as $N \in Q$, $\gamma(N_1) \leq \xi$.
  - $N_3 \notin P$ and $N_1 \xrightarrow{\xi}{}^{+}_{G_1} N_3$, so, as $N_3 \in R \setminus P$, we have $N_3 \in Q$ and $\gamma(N_1) \leq \xi$.
- $N_1 \xrightarrow{\xi_1}{}^{*}_{G\|R} N_2 \xrightarrow{\xi}_{G\|\{N_0\}} N_0$, with $N_1 \notin R$. By Property 10 again, as $N_0 \notin R$, we know that $N_1 \xrightarrow{\xi}{}^{+}_{G_1} N_0$, so $\gamma(N_1) \leq \xi$.

□

The next lemma deals with external merging (reduction rule EM). In fact, this reduction rule is also modeled through internalizing degree environments. For instance, consider the reduction step (`let rec` b_1 `in let rec` b_2 `in` e) $\longrightarrow$ (`let rec` b_1, b_2 `in` e). With respect to degree environments, the redex internalizes b_2 and then b_1, while the reduct internalizes b_1 and b_2 at once.

Lemma 10 (Two steps internalized degree environment)
Let P and Q be disjoint sets of nodes, and $N \notin P \cup Q$, and G be a dependency graph internalizable at P with entry point N, such that $\mathsf{Targets}(G) \subseteq P \cup Q \cup \{N\}$, *and* $\mathsf{Sources}(G) \# \{N\}$. *We have*

$$(\mathsf{Internalize}(G, P, N) \gg Q \gg N) \leq (G \gg (P \cup Q) \gg N).$$

Preuve Let $\gamma = (\mathsf{Internalize}(G, P, N) \gg Q \gg N)$ and $\gamma' = (G \gg P \cup Q \gg N)$. Let $G_1 = \mathsf{Internalize}(G, P, N)$. First, notice that the conditions on G imply that $\mathsf{Targets}(G_1) \subseteq Q \cup \{N\}$, and $\mathsf{Sources}(G_1) \# \{N\}$. Consequently, ${}_{Q\|}G_1 \subseteq G_{1\|Q \cup \{N\}}$. So the two degree environments are well-defined.

For any node N_1, let $\xi = \gamma'(N_1)$, and let $R = P \cup Q$. There are two possibilities, by definition of $(\cdot \gg \cdot \gg \cdot)$.

- $N_1 \xrightarrow{\xi_1}_{R-\|G\|R} N_2 \xrightarrow{\xi}{}^{*}_{R\|G\|R} N_3$ and $\xi \in \mathsf{Negative}$. We have $N_1 \notin P$, so by Property 10, there are two possibilities.
  - $N_3 \in P$ and $N_1 \xrightarrow{\xi}{}^{+}_{G_1} N$, so $\gamma(N_1) \leq \xi$.
  - $N_3 \notin P$ and $N_1 \xrightarrow{\xi}{}^{+}_{G_1} N_3$, so, as $N_3 \in R \setminus P$, we have $N_3 \in Q$ and $\gamma(N_1) \leq \xi$.

- $N_1 \xrightarrow{\xi_1}^*_{G_{\| R}} N_2 \xrightarrow{\xi}_{G_{\|\{N\}}} N$, with $N_1 \notin R$. By Property 10 again, as $N \notin R$, we know that $N_1 \xrightarrow{\xi}^+_{G_1} N$, so $\gamma(N_1) \leq \xi$.

$\square$

The next two results concern degree strengthening. The idea is to ensure that only the free variables matter in degree environments.

Proposition 11 (Degree environment restriction)
Let P and Q be disjoint finite sets of variables, a node $N \notin P \cup Q$, and G be a dependency graph internalizable at P with entry point N. We have $(G \gg P \gg N)_{|Q} = (({}_{P\cup Q\|}G) \gg P \gg N)$.

Preuve Let $G' = {}_{P\cup Q\|}G$, $\gamma = (G \gg P \gg N)_{|Q}$ and $\gamma' = (G' \gg P \gg N)$. First notice that $\mathsf{supp}(\gamma') \subseteq \mathsf{Sources}(G') \setminus P = \mathsf{Sources}(G) \cap Q \supseteq \mathsf{supp}(\gamma)$. So, for any node outside Q, both return ∞.

Then, let $G_1 = {}_{Q\|}G_{\|P}$. Let $G'_1 = {}_{Q\|}G'_{\|P}$. But $G' = {}_{P\cup Q\|}G$, so $G_1 = G'_1$.

Similarly, let $G_2 = {}_{P\|}G_{\|P}$ and $G'_2 = {}_{P\|}G'_{\|P}$. We have $G_2 = G'_2$.

Let $G_3 = {}_{P\|}G_{\|\{N\}}$ and $G'_3 = {}_{P\|}G'_{\|\{N\}}$. We have $G_3 = G'_3$.

Let $G_4 = {}_{Q\|}G_{\|\{N\}}$ and $G'_4 = {}_{Q\|}G'_{\|\{N\}}$. We have $G_4 = G'_4$.

Now, for all $N_0 \in Q$, $\gamma(N_0)$ is the minimum of the set of degrees ξ such that $N_0 \xrightarrow{\xi_1}_{G_1} N_1 \xrightarrow{\xi}^*_{G_2} N_2$ if ξ is negative, or $N_0 \xrightarrow{\xi_1}_{G_1} N_1 \xrightarrow{\xi_2}^*_{G_2} N_2 \xrightarrow{\xi}_{G_3} N$, or $N_0 \xrightarrow{\xi}_{G_4} N$. But, as we have seen, each of these paths corresponds to a path of the form $N_0 \xrightarrow{\xi_1}_{G'_1} N_1 \xrightarrow{\xi}^*_{G'_2} N_2$ or $N_0 \xrightarrow{\xi_1}_{G'_1} N_1 \xrightarrow{\xi_2}^*_{G'_2} N_2 \xrightarrow{\xi}_{G'_3} N$ or $N_0 \xrightarrow{\xi}_{G'_4} N$. So $\gamma'(N_0) \leq \gamma(N_0)$. The converse argument shows that $\gamma(N_0) \leq \gamma'(N_0)$, and therefore $\gamma(N_0) = \gamma'(N_0)$. $\square$

Corollary 1 (Graph / Degree restriction)
Let P and Q be disjoint finite sets of variables, $N_0 \notin P \cup Q$, and for all $N \in P \cup \{N_0\}$, γ_N be a degree environment. Let for each $N \in P \cup \{N_0\}$, $\gamma'_N = \gamma_{N\,|\,P\cup Q}$, $G = \bigcup_{N\in P\cup\{N_0\}} (\gamma_N \longrightarrow N)$, and $G' = \bigcup_{N\in P\cup\{N_0\}} (\gamma'_N \longrightarrow N)$. We have $(G \gg P \gg N_0)_{|Q} = (G' \gg P \gg N_0)$.

Then, we state some technical lemmas used to prove the subject reduction property. First, we give a sufficient condition for separating an internalized degree environment, in the following sense.

Proposition 12 (Graph separation)
Let G_1 and G_2 be two dependency graphs such that

- $N \notin \mathsf{Sources}(G_1 \cup G_2)$,
- $\mathsf{Targets}(G_1 \cup G_2) \subseteq P \uplus \{N\}$,
- $\mathsf{Targets}(G_1) \mathbin{\#} \mathsf{Sources}(G_2)$, *and*
- $\mathsf{Targets}(G_2) \mathbin{\#} \mathsf{Sources}(G_1)$.

We have $(G_1 \cup G_2 \gg P \gg N) = (G_1 \gg P \gg N) \wedge (G_2 \gg P \gg N)$.

Preuve First, the supports of both degree environments are included in $(\mathsf{Sources}(G_1) \cup \mathsf{Sources}(G_2)) \setminus P$. Now, let us fix a node N' in this set. The set $(G_1 \cup G_2)^+$ of paths of $G_1 \cup G_2$ is equal to the union $G_1^+ \cup G_2^+$, since no path can combine edges from both subgraphs. Furthermore, since $N \notin \mathsf{Sources}(G_1 \cup G_2)$, for any G in $\{(G_1 \cup G_2), G_1, G_2\}$, we have ${}_{P\cup\{N\}-\|}G^{+}{}_{\|P} = {}_{P-\|}((G_{\|P})^{+})$ and ${}_{P\cup\{N\}-\|}G^{+}{}_{\|\{N\}} = {}_{P-\|}((G_{\|P})^{*};G_{\|\{N\}})$. So, for any $N' \in \mathsf{supp}((G_1 \cup G_2 \gg P \gg N))$, $N' \notin P \cup \{N\}$, and

$$
\begin{aligned}
&((G_1 \cup G_2) \gg P \gg N)(N') = \\
&\qquad \mathsf{min}(\{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}{}^{+}_{(G_1\cup G_2)} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}{}^{+}_{(G_1\cup G_2)} N\}) \\
&= \mathsf{min}(\{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}_{(G_1^{+}\cup G_2^{+})} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}_{(G_1^{+}\cup G_2^{+})} N\}) \\
&= \mathsf{min}(\ \{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}_{G_1^{+}} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}_{G_1^{+}} N\} \cup \\
&\qquad\qquad \{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}_{G_2^{+}} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}_{G_2^{+}} N\}) \\
&= \mathsf{min}(\{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}_{G_1^{+}} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}_{G_1^{+}} N\}) \wedge \\
&\qquad \mathsf{min}(\{\chi^{\ominus} \mid N' \xrightarrow{\chi^{\ominus}}_{G_2^{+}} N'', N'' \in P\} \cup \{\xi \mid N' \xrightarrow{\xi}_{G_2^{+}} N\}) \\
&= (G_1 \gg P \gg N)(N') \wedge (G_2 \gg P \gg N)(N').
\end{aligned}
$$

□

The next property allows to ignore an irrelevant sub-graph when computing an internalized degree environment.

Proposition 13 (Graph irrelevance)
Let G_1 and G_2 be two dependency graphs such that $\mathsf{Targets}(G_2) \# P \cup \{N\}$ and ${}_{P\|}(G_1 \cup G_2) \subseteq (G_1 \cup G_2)_{\|P\cup\{N\}}$. We have $(G_1 \cup G_2 \gg P \gg N) = (G_1 \gg P \gg N)$.

Preuve Let $\gamma = (G_1 \cup G_2 \gg P \gg N)$ and $\gamma' = (G_1 \gg P \gg N)$. Notice that the hypotheses imply $\mathsf{Nodes}(G_2) \# P \cup \{N\}$. Obviously, we have $\gamma \leq \gamma'$. Now, assume $\xi = \gamma(N') \neq \infty$. There are two possibilities.

- $N' \xrightarrow{\xi}{}^{+}_{(G_1\cup G_2)_{\|P}} N''$, for some $N'' \in P$. But $(G_1 \cup G_2)_{\|P} = G_{1\|P}$, so $N' \xrightarrow{\xi}{}^{+}_{G_{1\|P}} N''$, and $\gamma'(N') \leq \xi$.
- $N' \xrightarrow{\xi_1}{}^{*}_{(G_1\cup G_2)_{\|P}} N_1 \xrightarrow{\xi}_{G_1\cup G_2} N$. But the last edge cannot be in G_2, so $N' \xrightarrow{\xi_1}{}^{*}_{G_{1\|P}} N_1 \xrightarrow{\xi}_{G_1} N$, and $\gamma'(N') \leq \xi$.

□

The next property factors the decrementing operation over an internalized degree environment.

Proposition 14 (Decrement)
Let P be a set of nodes, and $N \notin P$. For all $N' \in P \cup \{N\}$, let $\gamma_{N'}$ be a degree environment, such that $N \notin \mathsf{supp}(\gamma_{N'})$. We have

$$
(\bigcup_{N'\in P} (\gamma_{N'} \longrightarrow N') \cup ((\gamma_N - 1) \longrightarrow N) \gg P \gg N) \geq (\bigcup_{N'\in P\cup\{N\}} (\gamma_{N'} \longrightarrow N') \gg P \gg N) - 1.
$$

Preuve Let $G = \bigcup_{N'\in P\cup\{N\}} (\gamma_{N'} \longrightarrow N')$ and $G' = \bigcup_{N'\in P} (\gamma_{N'} \longrightarrow N') \cup ((\gamma_N - 1) \longrightarrow N)$. Let $\gamma = (G \gg P \gg N)$ and $\gamma' = (G' \gg P \gg N)$. We have to prove that $\gamma' \geq \gamma - 1$. Assume $N_1 \in \mathsf{supp}(\gamma')$, and let $\xi = \gamma'(N_1)$. We know that $N_1 \notin P$, and there are two possibilities:

- $N_1 \xrightarrow{\xi}{}^{+}_{G'_{\|P}} N_2$. But $G'_{\|P} = G_{\|P}$, so $\gamma(N_1) - 1 \leq \xi - 1$.
- $N_1 \xrightarrow{\xi_1}{}^{*}_{G'_{\|P}} N_2 \xrightarrow{\xi}_{G'_{\|\{N\}}} N$. Let $\xi' = \gamma_N(N_2)$. We have $\xi = \xi' - 1$, and $N_1 \xrightarrow{\xi_1}{}^{*}_{G_{\|P}} N_2 \xrightarrow{\xi'}_{G_{\|\{N\}}} N$, so $\gamma(N_1) - 1 \leq \xi' - 1 \leq \xi$.

□

The next property factors the apply operation over an internalized degree environment.

Proposition 15 (Apply)
Let P be a set of nodes, and $N \notin P$. For all $N' \in P \cup \{N\}$, let $\gamma_{N'}$ be a degree environment, such that $N \notin \mathsf{supp}(\gamma_{N'})$. We have

$$(\bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup ((\gamma_N \ominus 1) \longrightarrow N) \gg P \gg N) \geq (\bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N') \gg P \gg N) \ominus 1.$$

Preuve Let $G = \bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N')$ and $G' = \bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup ((\gamma_N \ominus 1) \longrightarrow N)$. Let $\gamma = (G \gg P \gg N)$ and $\gamma' = (G' \gg P \gg N)$. We have to prove that $\gamma' \geq \gamma \ominus 1$. Assume $N_1 \in \mathsf{supp}(\gamma')$, and let $\xi = \gamma'(N_1)$. We know that $N_1 \notin P$, and there are two possibilities:

- $N_1 \xrightarrow{\xi}{}^{+}_{G'_{\|P}} N_2$. But $G'_{\|P} = G_{\|P}$, so $\gamma(N_1) \ominus 1 \leq \xi \ominus 1 \leq \xi$.
- $N_1 \xrightarrow{\xi_1}{}^{*}_{G'_{\|P}} N_2 \xrightarrow{\xi}_{G'_{\|\{N\}}} N$. Let $\xi' = \gamma_N(N_2)$. We have $\xi = \xi' \ominus 1$, and $N_1 \xrightarrow{\xi_1}{}^{*}_{G_{\|P}} N_2 \xrightarrow{\xi'}_{G_{\|\{N\}}} N$, so $\gamma(N_1) \ominus 1 \leq \xi' \ominus 1 \leq \xi$.

□

The next property does the same with application to a fixed degree ξ.

Proposition 16
Let ξ be a generalized degree, P a set of nodes, and $N \notin P$. For all $N' \in P \cup \{N\}$, let $\gamma_{N'}$ be a degree environment, such that $N \notin \mathsf{supp}(\gamma_{N'})$. We have

$$(\bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup (\xi \mathbin{@} \gamma_N \longrightarrow N) \gg P \gg N) \geq \xi \mathbin{@} (\bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N') \gg P \gg N).$$

Preuve Let $G = \bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N')$ and $G' = \bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup (\xi \mathbin{@} \gamma_N \longrightarrow N)$. Let $\gamma = (G \gg P \gg N)$ and $\gamma' = (G' \gg P \gg N)$. Let $N_1 \in \mathsf{supp}(\gamma')$, with $\xi_1 = \gamma'(N_1)$. We know that $N_1 \notin P$, and there are two possibilities.

- If $N_1 \xrightarrow{-\infty}{}^{+}_{G'_{\|P}} N_2$, then obviously $\gamma(N_1) = -\infty$, so $\xi \mathbin{@} \gamma(N_1) = -\infty$.
- If $N_1 \xrightarrow{\xi_2}{}^{*}_{G'_{\|P}} N_2 \xrightarrow{\xi_1}_{G'} N$, with $\xi_1 = \xi \mathbin{@} \gamma_N(N_2)$. Then, $N_1 \xrightarrow{\xi_2}{}^{*}_{G_{\|P}} N_2 \xrightarrow{\gamma_N(N_2)}_{G} N$, so we have $\gamma(N_1) \leq \xi_1$, so by Lemma 7, $\xi \mathbin{@} \gamma(N_1) \leq \xi \mathbin{@} \xi_1$.

□

The next property does the same for the operation that lowers all the antecedents of 0 to -∞. This operation is implicitly used in rules T-LetRec and T-Select.

Proposition 17
Let P be a set of nodes, and $N \notin P$. For all $N' \in P \cup \{N\}$, let $\gamma_{N'}$ be a degree environment, such that $N \notin \mathsf{supp}(\gamma_{N'})$. Let also γ be a degree environment such that $\gamma \leq ((\bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N')) \gg P \gg N)$ and $\gamma^{-1}(0) = \emptyset$. Finally, let $\gamma'_N = \gamma_N \wedge -\infty_{|\gamma_N{}^{-1}(0)}$. We have

$$\gamma \leq ((\bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup (\gamma'_N \longrightarrow N)) \gg P \gg N).$$

Preuve Let $\gamma' = ((\bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N') \cup (\gamma'_N \longrightarrow N)) \gg P \gg N)$. For any node $N_1 \in \mathsf{supp}(\gamma')$, let $\xi = \gamma'(N_1)$. We have $N_1 \notin P$, and there are two possibilities.

- If $N_1 \xrightarrow{\xi}{}^{+}_{\| P} N_2$, then obviously $\gamma(N_1) \leq \xi$.
- If $N_1 \xrightarrow{\xi_1}{}^{*}_{\| P} N_2 \xrightarrow{\xi} N$, then:
  - If $\xi = \gamma_N(N_1)$, then obviously $\gamma(N_1) \leq \xi$.
  - Otherwise, $\gamma_N(N_1) = 0$, so $\gamma(N_1) < 0$, so $\gamma(N_1) = -\infty$.

□

Now, we examine a particular case of incrementing some edges in the graph unerlying a degree environment.

Proposition 18 (Increment)
Let $N \notin P$ and for all $N' \in P \cup \{N\}$, assume given a degree environment $\gamma_{N'}$ such that $N \notin \mathsf{supp}(\gamma_{N'})$. Let then $G = \bigcup_{N' \in P \cup \{N\}} (\gamma_{N'} \longrightarrow N')$ and $G' = ((\gamma_N + 1) \longrightarrow N) \cup \bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N')$.

We have

$$(G \gg P \gg N) + 1 \leq (G' \gg P \gg N).$$

Preuve Let $G_1 = \bigcup_{N' \in P} (\gamma_{N'} \longrightarrow N')$ and $G_2 = (\gamma_N + 1) \longrightarrow N$.

Let also $\gamma = (G \gg P \gg N) + 1$ and $\gamma' = (G' \gg P \gg N)$.

Let $N_1 \in \mathsf{supp}(\gamma')$. We must show that $\gamma(N_1) \leq \gamma'(N_1)$. So, we examine each path in G' that could contribute to $\gamma'(N_1)$, and find a path of inferior degree in G, which contributes to $\gamma(N_1)$.

There are two kinds of paths of G' contributing to $\gamma'(N_1)$.

- $N_1 \xrightarrow{\xi}{}^{+}_{G'_{\| P}} N_2$. But $G'_{\| P} = G_{\| P}$, and moreover if the path contributes to $\gamma'(N_1)$, then $\xi = -\infty$, so $\gamma(N_1) \leq \xi + 1 = -\infty = \xi$.
- $N_1 \xrightarrow{\xi_1}{}^{*}_{G'_{\| P}} N_2 \xrightarrow{\xi}_{G'_{\| \{N\}}} N$. Let $\xi' = \gamma_N(N_2)$. We have $\xi = \xi' + 1$, and $N_1 \xrightarrow{\xi_1}{}^{*}_{G_{\| P}} N_2 \xrightarrow{\xi'}_{G_{\| \{N\}}} N$, so $\gamma(N_1) \leq \xi' + 1 = \xi$.

□

Proposition 19 (Splitting)
Let N and N' be two nodes, P be a set of nodes, G be a dependency graph internalizable at P with entry point N, such that $N' \notin \mathsf{Targets}(G)$.

Then, let $G' = G_{N \succ N'}$. We have $(G' \gg P \gg N') = (G \gg P \gg N)$.

Preuve Let $\gamma = (G \gg P \gg N)$ and $\gamma' = (G' \gg P \gg N')$. Notice that $N, N' \notin P$, so $G_{\| P} = G'_{\| P}$.

- First, we prove $\gamma' \leq \gamma$. Let $N_1 \in \mathsf{supp}(\gamma)$. We know that $N_1 \notin P$. Let $\xi = \gamma(N_1)$. There are two possibilities.
  - $N_1 \xrightarrow{\xi}{}^{+}_{G_{\| P}} N_2$, with $N_2 \in P$. Then, this path is also a path of $G'_{\| P}$, so $\gamma'(N_1) \leq \xi$.
  - $N_1 \xrightarrow{\xi}{}^{*}_{G_{\| P}} N_2 \xrightarrow{\xi}_{G_{\| \{N\}}}$, with $N_2 \in P$. Then, by definition of splitting, $N_2 \xrightarrow{\xi}_{G'_{\| \{N'\}}} N'$, so $\gamma'(N_1) \leq \xi$.
- Then, we prove $\gamma \leq \gamma'$. Let $N_1 \in \mathsf{supp}(\gamma')$. We know that $N_1 \notin P$. Let $\xi = \gamma'(N_1)$. There are two possibilities.

– $N_1 \xrightarrow{\xi}^{+}_{G'_{\| P}} N_2$, with $N_2 \in P$. Then, this path is also a path of $G_{\| P}$, so $\gamma(N_1) \leq \xi$.

– $N_1 \xrightarrow{\xi}^{*}_{G'_{\| P}} N_2 \xrightarrow{\xi}_{G'_{\| \{N\}}}$, with $N_2 \in P$. Then, by definition of splitting, and as $N' \notin \mathsf{Targets}(G)$, we have $N_2 \xrightarrow{\xi}_{G_{\| \{N\}}} N$, so $\gamma(N_1) \leq \xi$.

□

Finally, we prove that a more restrictive dependency graph gives a more restrictive internalized degree environment.

Proposition 20 **($(\cdot \gg \cdot \gg \cdot)$ is monotone)**
If $G_1 \sqsubseteq G_2$, then $(G_1 \gg P \gg N) \leq (G_2 \gg P \gg N)$.

Preuve For all $N' \in \mathsf{supp}((G_2 \gg P \gg N))$, $(G_2 \gg P \gg N)(N') = \xi$ gives $N'' \in P \cup \{N\}$, and a path $N' \xrightarrow{\xi}^{+}_{G_2} N''$. But as $G_1 \sqsubseteq G_2$, there exists $\xi' \leq \xi$ such that $N' \xrightarrow{\xi'}^{+}_{G_2} N''$, so $(G_1 \gg P \gg N)(N') \leq (G_2 \gg P \gg N)(N')$. □

B Soundness

B.1 Weakening and strengthening lemmas

We first prove the weakening and strengthening lemmas stated in Sect. 5.

Lemma 1 (Type environment weakening) *If $\Gamma \vdash e : \tau \;/\; \gamma$, and $\mathsf{dom}(\Gamma') \# \mathsf{FV}(e)$, then $\Gamma + \Gamma' \vdash e : \tau \;/\; \gamma$.*

Lemma 2 (Type environment strengthening) *If $\Gamma + \Gamma' \vdash e : \tau \;/\; \gamma$, and $\mathsf{dom}(\Gamma') \# \mathsf{FV}(e)$, then $\Gamma \vdash e : \tau \;/\; \gamma$.*

Both lemmas are proved straightforwardly by induction on the derivation.

Lemma 3 (Degree environment weakening) *If $\gamma' \leq \gamma$ and $\Gamma \vdash e : \tau \;/\; \gamma$, then $\Gamma \vdash e : \tau \;/\; \gamma'$.*

Preuve We reason by induction on the typing derivation, and by case on the last typing rule used.

- Rules T-LetRec, T-Var, T-App, T-Select. By transitivity of $\leq$ on degrees.
- Rule T-Record. By induction hypothesis.
- Rule T-Abs, $e = \lambda x.e_1$. Given the typing rules, we have a derivation of $\Gamma + \{x : \tau_1\} \vdash e_1 : \tau_2 \;/\; (\gamma \ominus 1)\langle x \mapsto \xi\rangle$ with $\tau = \tau_1 \xrightarrow{\xi} \tau_2$. But by Lemma 7, $(\gamma' \ominus 1) \leq (\gamma \ominus 1)$, so $(\gamma' \ominus 1)\langle x \mapsto \xi\rangle \leq (\gamma \ominus 1)\langle x \mapsto \xi\rangle$, and so by induction hypothesis, we have a derivation of $\Gamma + \{x : \tau_1\} \vdash e_1 : \tau_2 \;/\; (\gamma' \ominus 1)\langle x \mapsto \xi\rangle$. The expected result follows by another application of rule T-Abs.

□

Lemma 4 (Degree environment strengthening) *If $\Gamma \vdash e : \tau \;/\; \gamma$, $P \supseteq \mathsf{FV}(e)$ and $\gamma'_{|P} = \gamma_{|P}$, then $\Gamma \vdash e : \tau \;/\; \gamma'$.*

Preuve By induction on the typing derivation of e and by case on the last rule used.

- Rule T-Var, $e = y$. Trivial.
- Rule T-Abs, $e = \lambda x.e_1$. By typing hypothesis,
  – $\Gamma + x : \tau_1 \vdash e_1 : \tau_2 \;/\; (\gamma \ominus 1)\langle x \mapsto \xi\rangle$,

– $\tau = \tau_1 \xrightarrow{\xi} \tau_2$.

We know that γ and γ' coincide on $P \supseteq \mathsf{FV}(e)$, so $(\gamma \ominus 1)\langle x \mapsto \xi\rangle$ and $(\gamma' \ominus 1)\langle x \mapsto \xi\rangle$ coincide on $P \cup \{x\} \supseteq \mathsf{FV}(e) \cup \{x\}$, so they coincide on $\mathsf{FV}(e_1) \subseteq \mathsf{FV}(e) \cup \{x\}$. By induction hypothesis, we deduce $\Gamma + \{x : \tau_1\} \vdash e_1 : \tau_2 \ / \ (\gamma' \ominus 1)\langle x \mapsto \xi\rangle$, so by rule T-Abs, $\Gamma \vdash e : \tau \ / \ \gamma'$.

- Rule T-App, $e = e_1\ e_2$. By typing hypothesis:
  - $\Gamma \vdash e_1 : \tau' \xrightarrow{\xi} \tau \ / \ \gamma_1$,
  - $\Gamma \vdash e_2 : \tau' \ / \ \gamma_2$,
  - $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2$.

  Let $\gamma'_1 = \gamma_{1|P}$ and $\gamma'_2 = \gamma_{2|P}$. By induction hypothesis, we derive $\Gamma \vdash e_1 : \tau' \xrightarrow{\xi} \tau \ / \ \gamma'_1$, and $\Gamma \vdash e_2 : \tau' \ / \ \gamma'_2$. So, in order to reconstruct the derivation for $\Gamma \vdash e : \tau \ / \ \gamma'$, we only have to show $\gamma' \leq ((\gamma'_1 \ominus 1) \wedge \xi \mathbin{@} \gamma'_2)$. But

$$\begin{aligned}(\gamma'_1 \ominus 1) \wedge \xi \mathbin{@} \gamma'_2 &= (\gamma_{1|P} \ominus 1) \wedge \xi \mathbin{@} \gamma_{2|P} \\ &= ((\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2)_{|P} \\ &\geq \gamma_{|P} \\ &= \gamma'_{|P} \\ &\geq \gamma'.\end{aligned}$$

- Rule T-LetRec, $e = \texttt{let rec}\ b\ \texttt{in}\ e_1$. By typing hypothesis, we have
  - a fresh res,
  - $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
  - for all $x \in \mathsf{dom}(b)$, $\Gamma + \Gamma_b \vdash b(x) : \Gamma_b(x) \ / \ \gamma_x$,
  - the conditions on sizes,
  - $\Gamma + \Gamma_b \vdash e_1 : \tau \ / \ \gamma_e$,
  - $G_b = \displaystyle\bigcup_{x \in \mathsf{dom}(b)} \gamma_x \longrightarrow x$,
  - $G = G_b \cup \gamma_e \longrightarrow res$,
  - $\vdash_{\lambda_\circ} (G_b, b)$,
  - $\gamma \leq (G \gg \mathsf{dom}(b) \gg res)$.

  The variable res can be chosen outside the support of γ'. Let $Q = \mathsf{dom}(b)$, with $\gamma_{res} = \gamma_e$. Let for each $x \in Q \cup \{res\}$, $\gamma'_x = \gamma_{x|Q \cup P}$, $G'_b = \displaystyle\bigcup_{x \in Q} \gamma'_x \longrightarrow x$, and $G' = G'_b \cup (\gamma'_e \longrightarrow res)$. We can see G as $\displaystyle\bigcup_{x \in Q \cup \{res\}} \gamma_x \longrightarrow x$.

  By induction hypothesis, we derive for all $x \in \mathsf{dom}(b)$, $\Gamma + \Gamma_b \vdash b(x) : \Gamma_b(x) \ / \ \gamma'_x$, and $\Gamma + \Gamma_b \vdash e_1 : \tau \ / \ \gamma'_{res}$. Moreover, $G'_b \subseteq G_b$, so $\vdash_{\lambda_\circ} (G'_b, b)$ holds. Furthermore, the conditions on sizes still hold. In particular, if $\diamond_x \neq =_{[?]}$, then $\gamma_x{}^{-1}(0) = \emptyset$, so $\gamma'_x{}^{-1}(0) = \gamma_{x|P \cup Q}{}^{-1} = \emptyset$. Finally, by Corollary 1, we have $(G \gg Q \gg res)_{|P} = (G' \gg Q \gg res)$. Moreover, by typing hypothesis, we have $\gamma_{|P} \leq (G \gg Q \gg res)_{|P}$, so $\gamma_{|P} \leq (G' \gg Q \gg res)$. But by hypothesis, $\gamma_{|P} = \gamma'_{|P}$, so $\gamma' \leq \gamma'_{|P} \leq (G' \gg Q \gg res)$, which gives the last premise needed to derive $\Gamma \vdash e : \tau \ / \ \gamma'$.

- Rule T-Record. We have a derivation of the shape

$$\frac{\mathsf{dom}(I) = \mathsf{dom}(s) \qquad \gamma \leq \gamma' + 1 \qquad \forall X \in \mathsf{dom}(s), \Gamma \vdash s(X) : I(X) \ / \ \gamma'}{\Gamma \vdash e : \tau \ / \ \gamma}$$

  with $\tau = \{I\}$ and $e = \{s\}$.

  By induction hypothesis, we derive for all $X \in \mathsf{dom}(s)$ $\Gamma \vdash s(X) : I(X) \ / \ \gamma'_{|P}$, so we get $\Gamma \vdash e : \tau \ / \ \gamma'_{|P} + 1$. But $\gamma'_{|P} + 1 = (\gamma' + 1)_{|P} \geq \gamma_{|P}$, which gives the expected result.

- Rule T-Select, $e = e_1.X$. We have a derivation of the shape

$$\frac{X \in \mathsf{dom}(I) \qquad \gamma \leq \gamma_1 - 1 \qquad \Gamma \vdash e_1 : \{I\} \;/\; \gamma_1}{\Gamma \vdash e : I(X) \;/\; \gamma}$$

By induction hypothesis, with $\gamma_1' = \gamma_{1\,|\,P}$, we derive $\Gamma \vdash e_1 : \{I\} \;/\; \gamma_1'$. Thus, we can apply rule T-Select again to obtain $\Gamma \vdash e : \tau \;/\; \gamma_1' - 1$. But then $\gamma_1' - 1 = \gamma_{1\,|\,P} - 1 = (\gamma_1 - 1)_{|\,P} \geq \gamma_{|\,P}$, which gives the expected result.

□

B.2 Subject reduction

We now examine the standard subject reduction property, which states that reduction preserves typing. We begin with a simple sufficient correctness condition in the presence of irrelevant subgraphs.

Proposition 21 (Binding correctness criterion)
Let b be a binding and $P = \mathsf{dom}(b)$. Let G_1 and G_2 be two dependency graphs such that $\mathsf{Targets}(G_2) \# \mathsf{Sources}(G_1) \cup P$. We have $\vdash_{\lambda_\circ} (G_1 \cup G_2, b)$ iff $\vdash_{\lambda_\circ} (G_1, b)$.

Preuve The predicate $\vdash_{\lambda_\circ} (G_1 \cup G_2, b)$ only concerns paths with ends in P, since it checks the compatibility of $(G^+)^{\text{-}\infty}$ and $\rightarrowtail_b^{pred}$ with $\rhd_b$. But: such a path cannot end with an edge of G_2 since $\mathsf{Targets}(G_2) \# P$, so it ends with an edge of G_1. However, as $\mathsf{Targets}(G_2) \# \mathsf{Sources}(G_1)$, no edge of G_1 can be preceded by an edge of G_2, so the concerned paths are all paths of G_1, and so $\vdash_{\lambda_\circ} (G_1 \cup G_2, b)$ is equivalent to $\vdash_{\lambda_\circ} (G_1, b)$. □

Next, we prove that contraction rules preserve typing.

Lemma 11 (Subject contraction)
If $e_1 \leadsto e_2$ and $\Gamma \vdash e_1 : \tau \;/\; \gamma$, then $\Gamma \vdash e_2 : \tau \;/\; \gamma$.

Preuve By case on the reduction rule.

- Rule Project, $e_1 = \{s\}.X$, $e_2 = s(X)$.

  We have a derivation of the shape

$$\frac{\dfrac{\forall Y \in \mathsf{dom}(s), \Gamma \vdash s(Y) : I(Y) \;/\; \gamma'' \qquad \gamma' \leq \gamma'' + 1}{\Gamma \vdash \{s\} : \{I\} \;/\; \gamma'} \qquad \gamma \leq \gamma' - 1}{\Gamma \vdash \{s\}.X : I(X) \;/\; \gamma}$$

  So, $\gamma \leq \gamma' - 1 \leq \gamma'' + 1 - 1 = \gamma''$, which gives the desired result.

- Rule Beta, $e_1 = (\lambda x.e)\ v$, $e_2 = \texttt{let rec}\ x =_{[?]} v\ \texttt{in}\ e$, $x \notin \mathsf{FV}(v)$. By typing, we have a derivation of the shape

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma + \{x : \tau_v\} \vdash e : \tau \;/\; (\gamma_1 \ominus 1)\langle x \mapsto \xi \rangle}}{\Gamma \vdash \lambda x.e : \tau_v \xrightarrow{\xi} \tau \;/\; \gamma_1} \qquad \dfrac{\vdots}{\Gamma \vdash v : \tau_v \;/\; \gamma_v}}{\Gamma \vdash e_1 : \tau \;/\; \gamma} \text{(T-App)}$$

  with $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_v$. As degree environments are of finite supports, we can assume w.l.o.g. that $x \notin \mathsf{supp}(\gamma_v)$ by α-conversion, and choose *res* fresh. Now, to reconstruct a typing derivation for e_2, let $\gamma_e = (\gamma_1 \ominus 1)\langle x \mapsto \xi \rangle$ and $G = (\gamma_v \longrightarrow x) \cup (\gamma_e \longrightarrow \mathit{res})$. First, we have $(\gamma_1 \ominus 1)_{\backslash x} = (\gamma_1 \ominus 1)\langle x \mapsto \infty \rangle = \gamma_{e \backslash x} \geq \gamma_1 \ominus 1$.

– If $\xi = \infty$, then we have $G^+ = G$, and

$$\begin{array}{rcl}(G \gg \{x\} \gg \mathit{res}) & = & \mathsf{if}_{\text{-}\infty}(\gamma_v) \wedge \gamma_e \\ & = & \mathsf{if}_{\text{-}\infty}(\gamma_v) \wedge \gamma_{e\backslash x} \\ & \geq & (\gamma_1 \ominus 1) \wedge \mathsf{if}_{\text{-}\infty}(\gamma_v) \\ & \geq & \gamma.\end{array}$$

– Otherwise, as $x, \mathit{res} \notin \mathsf{supp}(\gamma_v)$, we have $G^+ = G \cup \{y \xrightarrow{\xi} \mathit{res} \mid y \in \mathsf{supp}(\gamma_v)\}$. Therefore, $(G \gg \{x\} \gg \mathit{res}) = \mathsf{if}_{\text{-}\infty}(\gamma_v) \wedge \gamma_{e\backslash\{x\}} \wedge \xi_{|\,\mathsf{supp}(\gamma_v)} = \gamma_{e\backslash\{x\}} \wedge \mathsf{if}_{\text{-}\infty}(\gamma_v) \wedge (\xi\{\gamma_v\}) = \gamma_{e\backslash\{x\}} \wedge \xi \mathbin{@} \gamma_v$. But we have seen that $\gamma_{e\backslash x} \geq \gamma_1 \ominus 1$, so as $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_v$, we have $\gamma \leq \gamma_{e\backslash x} \wedge \xi \mathbin{@} \gamma_v$.

- Rule Lift. $e_1 = \mathbb{L}[\texttt{let rec}\ b\ \texttt{in}\ e_3]$, $e_2 = \texttt{let rec}\ b\ \texttt{in}\ \mathbb{L}[e_3]$, with $\mathsf{dom}(b) \mathbin{\#} \mathsf{FV}(\mathbb{L})$. By case on the lift context $\mathbb{L}$.

  – $\mathbb{L} = \Box\ e_4$. We have a typing derivation of the shape

$$\dfrac{\dfrac{\dfrac{\vdash_{\lambda_\circ} (G_b, b) \qquad \Gamma + \Gamma_b \vdash e_3 : \tau' \xrightarrow{\xi} \tau \mathbin{/} \gamma_{e_3}}{\Gamma + \Gamma_b \vdash b : \Gamma_b \mathbin{/} G_b}}{\Gamma \vdash \texttt{let rec}\ b\ \texttt{in}\ e_3 : \tau' \xrightarrow{\xi} \tau \mathbin{/} \gamma_1} \qquad \dfrac{\vdots}{\Gamma \vdash e_4 : \tau' \mathbin{/} \gamma_2}}{\Gamma \vdash (\texttt{let rec}\ b\ \texttt{in}\ e_3)\ e_4 : \tau \mathbin{/} \gamma}\ \text{(T-App)}$$

  with

  * res fresh,
  * $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
  * $G_b = \bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x)$,
  * $G = G_b \cup (\gamma_{e_3} \longrightarrow \mathit{res})$,
  * $\gamma_1 \leq (G \gg \mathsf{dom}(b) \gg \mathit{res})$,
  * and $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2$.

  By α-conversion, we can assume that $\mathsf{supp}(\gamma_2) \mathbin{\#} \mathsf{dom}(b) \cup \{\mathit{res}\}$. By Lemma 1, we can derive $\Gamma + \Gamma_b \vdash e_4 : \tau' \mathbin{/} \gamma_2$, and then by rule T-App, $\Gamma + \Gamma_b \vdash e_3\ e_4 : \tau \mathbin{/} \gamma'$, with $\gamma' = (\gamma_{e_3} \ominus 1) \wedge \xi \mathbin{@} \gamma_2$. So, in order to derive $\Gamma \vdash e_2 : \tau \mathbin{/} \gamma$, we let $G' = \bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x) \cup (\gamma' \longrightarrow \mathit{res})$.

  We only have to prove that $\gamma \leq (G' \gg \mathsf{dom}(b) \gg \mathit{res})$. For this, we partition G' as follows.

$$\begin{array}{rcl} G' & = & G_b \cup (((\gamma_{e_3} \ominus 1) \wedge \xi \mathbin{@} \gamma_2) \longrightarrow \mathit{res}) \\ & = & G_b \cup ((\gamma_{e_3} \ominus 1) \longrightarrow \mathit{res}) \cup (\xi \mathbin{@} \gamma_2 \longrightarrow \mathit{res}) \\ & = & G_3 \cup G_4, \end{array}$$

  with $G_3 = G_b \cup ((\gamma_{e_3} \ominus 1) \longrightarrow \mathit{res})$ and $G_4 = \xi \mathbin{@} \gamma_2 \longrightarrow \mathit{res}$. We have

  * $\mathit{res} \notin \mathsf{Sources}(G_3 \cup G_4)$,
  * $\mathsf{Targets}(G_3 \cup G_4) \subseteq \mathsf{dom}(b) \uplus \{\mathit{res}\}$,
  * $\mathsf{Targets}(G_3) \subseteq (\mathsf{dom}(b) \cup \{\mathit{res}\}) \mathbin{\#} \mathsf{supp}(\gamma_2) = \mathsf{Sources}(G_4)$,
  * and $\mathsf{Targets}(G_4) \subseteq \{\mathit{res}\} \mathbin{\#} \mathsf{Sources}(G_3)$.

  Therefore, we can apply Property 12, and obtain $(G' \gg \mathsf{dom}(b) \gg \mathit{res}) = (G_3 \gg \mathsf{dom}(b) \gg \mathit{res}) \wedge (G_4 \gg \mathsf{dom}(b) \gg \mathit{res})$. Obviously, we have $(G_4 \gg \mathsf{dom}(b) \gg \mathit{res}) = \xi \mathbin{@} \gamma_2$.

  Moreover, by Property 15, we have

$$(G_3 \gg \mathsf{dom}(b) \gg \mathit{res}) \geq (G \gg \mathsf{dom}(b) \gg \mathit{res}) \ominus 1.$$

Hence, by Lemma 7, we have

$$\begin{aligned}\gamma_1 \ominus 1 &\leq (G \gg \mathsf{dom}(b) \gg \mathit{res}) \ominus 1\\ &\leq (G_3 \gg \mathsf{dom}(b) \gg \mathit{res}).\end{aligned}$$

So,

$$\begin{aligned}(\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2 &\leq (G_3 \gg \mathsf{dom}(b) \gg \mathit{res}) \wedge \xi \mathbin{@} \gamma_2\\ &\leq (G_3 \gg \mathsf{dom}(b) \gg \mathit{res}) \wedge (G_4 \gg \mathsf{dom}(b) \gg \mathit{res})\\ &\leq (G' \gg \mathsf{dom}(b) \gg \mathit{res}).\end{aligned}$$

So, putting it all together:

$$\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2 \leq (G' \gg \mathsf{dom}(b) \gg \mathit{res}),$$

which gives the desired result.

– $\mathbb{L} = v\ \square$. We have a typing derivation of the shape

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash v : \tau' \xrightarrow{\xi} \tau \ /\ \gamma_1} \qquad \dfrac{\dfrac{\Gamma + \Gamma_b \vdash e_3 : \tau' \ /\ \gamma_{e_3}}{\Gamma_b + \Gamma_b \vdash b : \Gamma_b \ /\ G_b \qquad \vdash_{\lambda_\circ} (G_b, b)}}{\Gamma \vdash \texttt{let rec}\ b\ \texttt{in}\ e_3 : \tau' \ /\ \gamma_2}}{\Gamma \vdash v\ (\texttt{let rec}\ b\ \texttt{in}\ e_3) : \tau \ /\ \gamma} \ (\text{T-App})$$

with

* res fresh,
* $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
* $G_b = \bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x)$,
* $G = G_b \cup (\gamma_{e_3} \longrightarrow \mathit{res})$,
* $\gamma_2 \leq (G \gg \mathsf{dom}(b) \gg \mathit{res})$,
* and $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_2$.

By α-conversion, we can assume that $\mathsf{supp}(\gamma_1) \mathbin{\#} \mathsf{dom}(b) \cup \{\mathit{fresh}\}$. By Lemma 1, we derive $\Gamma + \Gamma_b \vdash v : \tau' \xrightarrow{\xi} \tau \ /\ \gamma_1$. So, by rule T-App, $\Gamma + \Gamma_b \vdash v\ e_3 : \tau \ /\ \gamma'$, with $\gamma' = (\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_{e_3}$.

So, in order to derive $\Gamma \vdash e_2 : \tau \ /\ \gamma$, we let $G' = G_b \cup (\gamma' \longrightarrow \mathit{res})$, and we only have to prove that $\gamma \leq (G' \gg \mathsf{dom}(b) \gg \mathit{res})$. For this, let $G_3 = G_b \cup (\xi \mathbin{@} \gamma_{e_3} \longrightarrow \mathit{res})$ and $G_4 = (\gamma_1 \ominus 1) \longrightarrow \mathit{res}$. We have $G' = G_b \cup (((\gamma_1 \ominus 1) \wedge \xi \mathbin{@} \gamma_{e_3}) \longrightarrow \mathit{res}) = G_3 \cup G_4$. By Property 12, as $\mathit{res} \notin \mathsf{Sources}(G')$, $\mathsf{Targets}(G') \subseteq \mathsf{dom}(b) \cup \{\mathit{res}\}$, $\mathsf{Targets}(G_3) \subseteq \mathsf{dom}(b) \cup \{\mathit{res}\} \mathbin{\#} \mathsf{Sources}(G_4) = \mathsf{supp}(\gamma_1)$, and $\mathsf{Targets}(G_4) \subseteq \{\mathit{res}\} \mathbin{\#} \mathsf{Sources}(G_3)$, we have $(G' \gg \mathsf{dom}(b) \gg \mathit{res}) = (G_3 \gg \mathsf{dom}(b) \gg \mathit{res}) \wedge (G_4 \gg \mathsf{dom}(b) \gg \mathit{res})$.

Let $\gamma_0 = (G_3 \gg \mathsf{dom}(b) \gg \mathit{res})$, and notice $(G_4 \gg \mathsf{dom}(b) \gg \mathit{res}) = \gamma_1 \ominus 1$. Thus, we just have to prove $\gamma_0 \geq \xi \mathbin{@} \gamma_2$. But by Property 16, we have $\gamma_0 \geq \xi \mathbin{@} (G \gg \mathsf{dom}(b) \gg \mathit{res})$, which gives the desired result by Lemma 7.

– $\mathbb{L} = \square.X$. We have a typing derivation of the shape,

$$\dfrac{\gamma \leq \gamma_1 - 1 \qquad X \in \mathsf{dom}(I) \qquad \dfrac{\Gamma' \vdash e_3 : \{I\} \ /\ \gamma_{e_3} \qquad \Gamma' \vdash b : \Gamma_b \ /\ G_b \qquad \vdash_{\lambda_\circ} (G_b, b)}{\Gamma \vdash \texttt{let rec}\ b\ \texttt{in}\ e_3 : \{I\} \ /\ \gamma_1}}{\Gamma \vdash e_1 : I(X) \ /\ \gamma}$$

with

* res fresh,
* $\Gamma' = \Gamma + \Gamma_b$,
* $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
* $G = G_b \cup (\gamma_{e_3} \longrightarrow \mathit{res})$,
* and $\gamma_1 \leq (G \gg \mathsf{dom}(b) \gg \mathit{res})$.

We can reconstruct

$$\dfrac{\dfrac{\Gamma' \vdash e_3 : \{I\} \ / \ \gamma_{e_3}}{\Gamma' \vdash e_3.X : I(X) \ / \ (\gamma_{e_3} - 1)} \qquad \Gamma' \vdash b : \Gamma_b \ / \ G_b \qquad \vdash_{\lambda_\circ} (G_b, b)}{\Gamma' \vdash e' : I(X) \ / \ \gamma'}$$

where

* $\gamma' = (G' \gg \mathsf{dom}(b) \gg \mathit{res})$,
* and $G' = G_b \cup ((\gamma_{e_3} - 1) \longrightarrow \mathit{res})$.

We then just have to prove that $\gamma' \geq \gamma$.

But $\gamma \leq \gamma_1 - 1 \leq (G \gg \mathsf{dom}(b) \gg \mathit{res}) - 1$, which by Prop. 14 is inferior to $((G_b \cup ((\gamma_{e_3} - 1) \longrightarrow \mathit{res})) \gg \mathsf{dom}(b) \gg \mathit{res})$, which is exactly γ'.

□

Now, we turn to proving that the global reduction rules preserve typing. We begin with some results on internal merging.

Proposition 22 (Internal merging)
If $e_1 \overset{\text{IM}}{\longrightarrow} e_2$ *and* $\Gamma \vdash e_1 : \tau \ / \ \gamma$, *then* $\Gamma \vdash e_2 : \tau \ / \ \gamma$.

Preuve We know that $e_1 = \texttt{let rec}\ b_v, x_0 \diamond (\texttt{let rec}\ b_1\ \texttt{in}\ e_3), b_2\ \texttt{in}\ e_4$ and $e_2 = \texttt{let rec}\ b_v, b_1, x_0 \diamond e_3, b_2\ \texttt{in}\ e_4$. Let $b = (b_v, x_0 \diamond (\texttt{let rec}\ b_1\ \texttt{in}\ e_3), b_2)$ and $b' = (b_v, b_1, x_0 \diamond e_3, b_2)$. We know that $\mathsf{dom}(b_1) \# \mathsf{dom}(b) \cup \mathsf{FV}(b) \cup \mathsf{FV}(e_4)$. We have a typing derivation of the shape

$$\dfrac{\Gamma + \Gamma_b \vdash e_4 : \tau \ / \ \gamma_{e_4} \qquad \Gamma + \Gamma_b \vdash b : \Gamma_b \ / \ G_b \qquad \vdash_{\lambda_\circ} (G_b, b)}{\Gamma \vdash e_1 : \tau \ / \ \gamma}$$

with

- res fresh,
- $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
- $G_b = \bigcup_{x \in \mathsf{dom}(b)} (\gamma_x \longrightarrow x)$,
- $G = G_b \cup (\gamma_{e_4} \longrightarrow \mathit{res})$,
- and $\gamma \leq (G \gg \mathsf{dom}(b) \gg \mathit{res})$.

By α-conversion, we can assume $\mathsf{supp}(\gamma) \# \mathsf{dom}(b) \cup \{\mathit{res}\} \cup \mathsf{dom}(b_1)$. For $x_0 \diamond \texttt{let rec}\ b_1\ \texttt{in}\ e_3$, we have

$$\dfrac{\Gamma + \Gamma_b + \Gamma_{b_1} \vdash e_3 : \Gamma_b(x_0) \ / \ \gamma_{e_3} \qquad \Gamma + \Gamma_b + \Gamma_{b_1} \vdash b_1 : \Gamma_{b_1} \ / \ G_{b_1} \qquad \vdash_{\lambda_\circ} (G_0, b_1)}{\Gamma \vdash \texttt{let rec}\ b_1\ \texttt{in}\ e_3 : \Gamma_b(x_0) \ / \ \gamma_{x_0}}$$

with

- res_0 fresh,
- $\mathsf{dom}(\Gamma_{b_1}) = \mathsf{dom}(b_1)$,
- $G_{b_1} = \bigcup_{y \in \mathsf{dom}(b_1)} (\gamma_y \longrightarrow y)$,
- $G_0 = G_{b_1} \cup (\gamma_{e_3} \longrightarrow \mathit{res}_0)$,

- and $\gamma_{x_0} \leq (G_0 \gg \mathsf{dom}(b_1) \gg res_0)$.

Let $G'_b = G_{b\|-\{x_0\}} \cup (\gamma_{e_3} \longrightarrow x_0) \cup G_{b_1}$ and $G' = G'_b \cup (\gamma_{e_4} \longrightarrow res)$.

By weakening, we obtain $\Gamma + \Gamma_b + \Gamma_{b_1} \vdash e_x : \Gamma_b(x) \;/\; \gamma_x$ for all $x \in \mathsf{dom}(b) \setminus \{x_0\}$. Further, we know that for all $x \in \mathsf{dom}(b') \setminus \{x_0\}$, if $\diamond_x \neq =_{[?]}$, then ${\gamma_x}^{-1}(0) = \emptyset$. For x_0, if $\diamond_{x_0} = =_{[?]}$, then let $\gamma'_{e_3} = \gamma_{e_3} \wedge \text{-}\infty_{|\,{\gamma_{e_3}}^{-1}(0)}$. By degree weakening, we have $\Gamma + \Gamma_b + \Gamma_{b_1} \vdash e_3 : \Gamma_b(x_0) \;/\; \gamma'_{e_3}$, and by Property 17, we obtain $\gamma_{x_0} \leq (G_{b_1} \cup (\gamma'_{e_3} \longrightarrow res_0) \gg \mathsf{dom}(b_1) \gg res_0)$. So, w.l.o.g., we can assume that ${\gamma_{e_3}}^{-1}(0) = \emptyset$.

So, in order to derive $\Gamma \vdash e_2 : \tau \;/\; \gamma$, we just have to prove that $\gamma \leq (G' \gg \mathsf{dom}(b') \gg res)$ and $\vdash_{\lambda_\circ} (G'_b, b')$.

Let $G'_0 = G_{b_1} \cup (\gamma_{e_3} \longrightarrow x_0)$, $\gamma_0 = (G_0 \gg \mathsf{dom}(b_1) \gg res_0)$, and $\gamma'_0 = (G'_0 \gg \mathsf{dom}(b_1) \gg x_0)$. Now, let $G_1 = \bigcup_{x \in \mathsf{dom}(b)\setminus\{x_0\}} (\gamma_x \longrightarrow x) \cup (\gamma_0 \longrightarrow x_0) \cup (\gamma_{e_4} \longrightarrow res)$ and $G_2 = \bigcup_{x \in \mathsf{dom}(b)\setminus\{x_0\}} (\gamma_x \longrightarrow x) \cup (\gamma'_0 \longrightarrow x_0) \cup (\gamma_{e_4} \longrightarrow res)$.

But we have $G_0 = G_{0\,x_0 \succ res_0}$, so by Property 19, $\gamma_0 = \gamma'_0$, so $G_1 = G_2$.

Now, let us examine $(G' \gg \mathsf{dom}(b_1) \gg x_0)$. Let $G_3 = \bigcup_{x \in \mathsf{dom}(b)\setminus\{x_0\}} (\gamma_x \longrightarrow x) \cup (\gamma_{e_4} \longrightarrow res)$. We have $G' = G'_0 \cup G_3$, and $\mathsf{Targets}(G_3) \# \mathsf{dom}(b_1) \cup x_0$, so by Property 13, $(G' \gg \mathsf{dom}(b_1) \gg x_0) = (G'_0 \gg \mathsf{dom}(b_1) \gg x_0) = \gamma'_0$. Further, $G_3 = G'_{\|-\mathsf{dom}(b_1)\cup\{x_0\}} = {}_{\mathsf{dom}(b_1)-\|}G'_{\|-\mathsf{dom}(b_1)\cup\{x_0\}}$, so as $G_2 = G_3 \cup (\gamma'_0 \longrightarrow x_o)$, we have $G_2 = \mathsf{Internalize}(G', \mathsf{dom}(b_1), x_0)$. Thus, we can apply Lemma 9 to obtain $(G_2 \gg \mathsf{dom}(b) \gg res) \leq (G' \gg \mathsf{dom}(b') \gg res)$.

But $G_1 = G_2$, so $(G_1 \gg \mathsf{dom}(b) \gg res) = (G_2 \gg \mathsf{dom}(b) \gg res)$. Further, we know that $\gamma_{x_0} \leq \gamma_0$, thus $G \sqsubseteq G_1$, so $\gamma \leq (G \gg \mathsf{dom}(b) \gg res) \leq (G_1 \gg \mathsf{dom}(b) \gg res)$, so finally $\gamma \leq (G' \gg \mathsf{dom}(b') \gg res)$.

Now we prove $\vdash_{\lambda_\circ} (G'_b, b')$. We have $\gamma_{x_0} \leq \gamma'_0$, so $G_b \sqsubseteq G_{b\|-\{x_0\}} \cup (\gamma'_0 \longrightarrow x_0) = \mathsf{Internalize}(G'_b, \mathsf{dom}(b_1), x_0)$. But $\vdash (G_b, \rhd_b)$, so $\vdash (\mathsf{Internalize}(G'_b, \mathsf{dom}(b_1), x_0), \rhd_b)$. Further, as the nodes of this graph are not in $\mathsf{dom}(b_1)$, we also have $\vdash (\mathsf{Internalize}(G'_b, \mathsf{dom}(b_1), x_0), \rhd_{b'})$. Moreover, ${}_{\mathsf{dom}(b_1)\|}G'_{b\|\,\mathsf{dom}(b_1)} \subseteq {}_{\mathsf{dom}(b_1)\|}G_{b_1\|\,\mathsf{dom}(b_1)} \subseteq G_{b_1}$. So, since $\vdash (G_{b_1}, \rhd_{b'})$, we obtain $\vdash ({}_{\mathsf{dom}(b_1)\|}G'_{b\|\,\mathsf{dom}(b_1)}, \rhd_{b_1})$. Thus, we can apply Lemma 8 to obtain $\vdash (G'_b, b')$. Finally, $\vdash (\rightarrowtail^{pred}_{b'}, \rhd_{b'})$ follows from the shape of b' and $\vdash (\rightarrowtail^{pred}_{b}, \rhd_b)$ and $\vdash (\rightarrowtail^{pred}_{b_1}, \rhd_{b_1})$.
□

We continue with external merging.

Proposition 23 (External merging)
If $e_1 \overset{\text{EM}}{\longrightarrow} e_2$ *and* $\Gamma \vdash e_1 : \tau \;/\; \gamma$*, then* $\Gamma \vdash e_2 : \tau \;/\; \gamma$*.*

Preuve We know that $e_1 = \texttt{let rec } b_v \texttt{ in let rec } b \texttt{ in } e_3$ and $e_2 = \texttt{let rec } b_v, b \texttt{ in } e_3$. We have a typing derivation of the shape

$$\frac{\Gamma + \Gamma_{b_v} \vdash \texttt{let rec } b \texttt{ in } e_3 : \tau \;/\; \gamma_b \qquad \Gamma + \Gamma_{b_v} \vdash b_v : \Gamma_{b_v} \;/\; G_{b_v} \qquad \vdash_{\lambda_\circ} (G_{b_v}, b_v)}{\Gamma \vdash e_1 : \tau \;/\; \gamma}$$

with

- res fresh,
- $\mathsf{dom}(\Gamma_{b_v}) = \mathsf{dom}(b_v)$,
- $G_{b_v} = \bigcup_{x \in \mathsf{dom}(b_v)} (\gamma_x \longrightarrow x)$,
- $G_v = G_{b_v} \cup (\gamma_b \longrightarrow res)$,
- and $\gamma \leq (G_v \gg \mathsf{dom}(b_v) \gg res)$.

For $\Gamma + \Gamma_{b_v} \vdash \texttt{let rec } b \texttt{ in } e_3 : \tau \;/\; \gamma_b$, we have

$$\frac{\Gamma + \Gamma_{b_v} + \Gamma_b \vdash e_3 : \tau \;/\; \gamma_{e_3} \qquad \Gamma + \Gamma_{b_v} + \Gamma_b \vdash b : \Gamma_b \;/\; G_b \qquad \vdash_{\lambda_\circ} (G_b, b)}{\Gamma \vdash \texttt{let rec } b \texttt{ in } e_3 : \tau \;/\; \gamma_b}$$

with

- res_0 fresh,
- $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
- $G_b = \bigcup_{y \in \mathsf{dom}(b)} (\gamma_y \longrightarrow y)$,
- $G = G_b \cup (\gamma_{e_3} \longrightarrow res_0)$,
- and $\gamma_b \leq (G_b \gg \mathsf{dom}(b) \gg res_0)$.

Let $b' = b_v, b$ and $G'_b = G_{b_v} \cup G_b$, and $G' = G_b \cup (\gamma_{e_3} \longrightarrow res_0)$. The only non trivial points to prove $\Gamma \vdash e_2 : \tau \;/\; \gamma$ are to prove that $\vdash_{\lambda_\circ} (G'_b, b')$ and $\gamma \leq (G' \gg \mathsf{dom}(b') \gg res_0)$.

1. For the first point, we obtain $\vdash (\rightarrowtail^{pred}_{b'}, \rhd_{b'})$ from the shape of b' and $\vdash (\rightarrowtail^{pred}_{b_v}, \rhd_{b_v})$ and $\vdash (\rightarrowtail^{pred}_{b}, \rhd_{b})$. Further, by Property 23, we get $\vdash (G_1 \cup G_2, \rhd_{b'})$, which gives the desired result.

2. For the second point, it is enough to prove that $(G_v \gg \mathsf{dom}(b_v) \gg res) \leq (G' \gg \mathsf{dom}(b') \gg res_0)$. Let $\gamma'_b = (G_b \gg \mathsf{dom}(b) \gg res_0)$, $G'_v = G_1 \cup (\gamma'_b \longrightarrow res)$, and $G''_v = G_1 \cup (\gamma'_b \longrightarrow res_0)$.

   We have immediately $(G'_v \gg \mathsf{dom}(b_v) \gg res) = (G''_v \gg \mathsf{dom}(b_v) \gg res_0)$, and $G_v \sqsubseteq G'_v$, which implies $(G_v \gg \mathsf{dom}(b_v) \gg res) \leq (G'_v \gg \mathsf{dom}(b_v) \gg res)$, and therefore $(G_v \gg \mathsf{dom}(b_v) \gg res) \leq (G''_v \gg \mathsf{dom}(b_v) \gg res_0)$. So, we only have to prove $(G''_v \gg \mathsf{dom}(b_v) \gg res_0) \leq (G' \gg \mathsf{dom}(b') \gg res_0)$.

   Now, we have ${}_{\mathsf{dom}(b)-\|}G'_{\|-\mathsf{dom}(b)\cup\{res_0\}} = G_1$ and $G' = G_1 \cup G_b$, with ${}_{\mathsf{dom}(b)\|}G' \subseteq G'_{\|\,\mathsf{dom}(b)\cup\{res_0\}}$ and $\mathsf{Targets}(G_1) \;\#\; \mathsf{dom}(b) \cup \{res_0\}$, so by Property 13, $(G' \gg \mathsf{dom}(b) \gg res_0) = (G_b \gg \mathsf{dom}(b) \gg res_0) = \gamma'_b$. So, $\mathsf{Internalize}(G', \mathsf{dom}(b), res_0) = G_1 \cup (\gamma'_b \longrightarrow res_0) = G''_v$.

   So, by Lemma 10, $(G' \gg \mathsf{dom}(b') \gg res_0) = (\mathsf{Internalize}(G', \mathsf{dom}(b), res_0) \gg \mathsf{dom}(b_v) \gg res_0) = (G''_v \gg \mathsf{dom}(b_v) \gg res_0)$.

□

Next, we examine rule Context. We prove that typing is compositional, which entails the desired result, in combination with Lemma 11.

Proposition 24 (Typing is compositional)
Assume given an expression e and an evaluation context $\mathbb{E}$, such that $\Gamma \vdash \mathbb{E}[e] : \tau \;/\; \gamma$, with a sub-derivation $\Gamma' \vdash e : \tau' \;/\; \gamma'$ for e. If $\Gamma' \vdash e' : \tau' \;/\; \gamma'$ and $\mathsf{FV}(e') \subseteq \mathsf{FV}(e)$, then $\Gamma \vdash \mathbb{E}[e'] : \tau \;/\; \gamma$.

Preuve Simple induction on the typing derivation of $\mathbb{E}[e]$. The condition on free variables preserves the well-formedness of bindings w.r.t. backward dependencies on definitions of unknown sizes. □

Finally, we turn to rule Subst. We distinguish internal substitution (access by rule IA) from external substitution (access by rule EA). The proof for rule EA is not too difficult, but the one for rule IA is harder. By application of this rule a binding of the shape $b_1 = (b_v, y \diamond \mathbb{F}[\mathbb{A}[x]], b)$ becomes $b_2 = (b_v, y \diamond \mathbb{F}[\mathbb{A}[v]], b)$, where $v = b_v(x)$. The idea is that the dependency graph of b_2 is less restrictive than the one of b_1, because for each edge induced by v in b_2, there is an equivalent path through x in b_1. Thus, after internal merging we obtain an greater degree environment, which is safe.

Proposition 25 (Dereferencing context)
If $\Gamma \vdash v : \tau_v \,/\, \gamma_v$, $\Gamma(x) = \tau_v$, and $\Gamma \vdash \mathbb{A}\,[x] : \tau \,/\, \gamma$, then $\gamma(x) = \text{-}\infty$ and there exists γ'_v such that $\mathsf{supp}(\gamma'_v) \subseteq \mathsf{supp}(\gamma_v)$ and $\Gamma \vdash \mathbb{A}\,[v] : \tau \,/\, \gamma \wedge \gamma'_v$.

Preuve By case on $\mathbb{A}$.

- $\mathbb{A} = \Box\ v_1$. We have a derivation of the shape

$$\frac{\Gamma(x) = \tau_1 \xrightarrow{\xi} \tau = \tau_v \qquad \Gamma \vdash v_1 : \tau_1 \,/\, \gamma_2 \qquad \gamma_1(x) = \text{-}\infty}{\Gamma \vdash \mathbb{A}\,[x] : \tau \,/\, \gamma}$$

  with $\gamma \leq (\gamma_1 \ominus 1) \wedge \xi\,@\,\gamma_2$ for some γ_1. We have immediately $\gamma(x) = \text{-}\infty$. Let $\gamma'_v = \gamma_v \ominus 1$. We have $\Gamma \vdash v : \tau_1 \xrightarrow{\xi} \tau \,/\, \gamma_v$, and by Lemma 3, we obtain $\Gamma \vdash v : \tau_1 \xrightarrow{\xi} \tau \,/\, \gamma_v \wedge \gamma_1$. So, we derive

$$\frac{\Gamma \vdash v_1 : \tau_1 \,/\, \gamma_2 \qquad \Gamma \vdash v : \tau_1 \xrightarrow{\xi} \tau \,/\, \gamma_v \wedge \gamma_1}{\Gamma \vdash \mathbb{A}\,[v] : \tau \,/\, \gamma \wedge \gamma'_v}$$

  since $\gamma \wedge \gamma'_v = \gamma \wedge (\gamma_v \ominus 1) \leq (\gamma_1 \ominus 1) \wedge \xi\,@\,\gamma_2 \wedge (\gamma_v \ominus 1) = (\gamma_v \wedge \gamma_1 \ominus 1) \wedge \xi\,@\,\gamma_2$. Finally, $\mathsf{supp}(\gamma'_v) = \mathsf{supp}(\gamma_v)$.

- $\mathbb{A} = \Box.X$. We have a derivation of the shape

$$\frac{X \in \mathsf{dom}(I) \qquad \gamma \leq \gamma_1 - 1 \qquad \Gamma(x) = \tau_v = \{I\}}{\Gamma \vdash \mathbb{A}\,[x] : I(X) \,/\, \gamma}$$

  with $\gamma_1(x) \leq 0$, so $\gamma(x) = \text{-}\infty$. Let $\gamma'_v = \gamma_v - 1$ and $\gamma' = \gamma \wedge \gamma'_v$. The degree environment γ'_v has the same support as γ_v and is such that $\gamma' \leq \gamma$. We have $\Gamma \vdash v : \{I\} \,/\, \gamma_v$, so by Lemma 3, we obtain $\Gamma \vdash v : \{I\} \,/\, \gamma_v \wedge \gamma_1$. So we can derive

$$\frac{X \in \mathsf{dom}(I) \qquad \Gamma \vdash v : \tau_v \,/\, (\gamma_1 \wedge \gamma_v)}{\Gamma \vdash \mathbb{A}\,[v] : I(X) \,/\, (\gamma_1 \wedge \gamma_v) - 1}$$

  Finally, we prove that $\gamma' = \gamma \wedge \gamma'_v \leq (\gamma_1 - 1) \wedge (\gamma_v - 1) = (\gamma_1 \wedge \gamma_v) - 1$, so we derive $\Gamma \vdash \mathbb{A}\,[v] : I(X) \,/\, \gamma'$.

□

Proposition 26 (Lift context)
Assume $\Gamma \vdash e : \tau \,/\, \gamma$, $\Gamma \vdash e' : \tau \,/\, \gamma'$, such that $\gamma' = \gamma \wedge \gamma_v$ for some γ_v. If $\Gamma \vdash \mathbb{L}\,[e] : \tau_{\mathbb{L}} \,/\, \gamma_{\mathbb{L}}$, then there exists γ'_v, with $\mathsf{supp}(\gamma'_v) \subseteq \mathsf{supp}(\gamma_v)$, such that $\Gamma \vdash \mathbb{L}\,[e'] : \tau_{\mathbb{L}} \,/\, (\gamma_{\mathbb{L}} \wedge \gamma'_v)$. Moreover, $\gamma_{\mathbb{L}} \leq \mathsf{if}_{\text{-}\infty}(\gamma)$.

Preuve By case on $\mathbb{L}$.

- $\mathbb{L} = v\ \Box$. We have a derivation of the shape

$$\frac{\Gamma \vdash v : \tau \xrightarrow{\xi} \tau_{\mathbb{L}} \,/\, \gamma_1 \qquad \Gamma \vdash e : \tau \,/\, \gamma \qquad \gamma_{\mathbb{L}} \leq (\gamma_1 \ominus 1) \wedge \xi\,@\,\gamma}{\Gamma \vdash \mathbb{L}\,[e] : \tau_{\mathbb{L}} \,/\, \gamma_{\mathbb{L}}}$$

  First, $\gamma_{\mathbb{L}} \leq \xi\,@\,\gamma \leq \mathsf{if}_{\text{-}\infty}(\gamma)$. Then, let $\gamma'_v = \xi\,@\,\gamma_v$. By Lemma 7, $\xi\,@\,(\gamma \wedge \gamma_v) = \xi\,@\,\gamma \wedge \xi\,@\,\gamma_v$. So, we obviously can derive $\Gamma \vdash \mathbb{L}\,[e'] : \tau_{\mathbb{L}} \,/\, (\gamma_{\mathbb{L}} \wedge \gamma'_v)$. We have $\mathsf{supp}(\gamma'_v) \subseteq \mathsf{supp}(\gamma_v)$.

- $\mathbb{L} = \Box\ e_1$. The derivation has the shape

$$\frac{\Gamma \vdash e : \tau_1 \xrightarrow{\xi} \tau_{\mathbb{L}} \,/\, \gamma \qquad \Gamma \vdash e_1 : \tau_1 \,/\, \gamma_1 \qquad \gamma_{\mathbb{L}} \leq (\gamma \ominus 1) \wedge \xi\,@\,\gamma_1}{\Gamma \vdash \mathbb{L}\,[e] : \tau_{\mathbb{L}} \,/\, \gamma_{\mathbb{L}}}$$

  with $\tau = \tau_1 \xrightarrow{\xi} \tau_{\mathbb{L}}$. First, $\gamma_{\mathbb{L}} \leq (\gamma \ominus 1) \leq \mathsf{if}_{\text{-}\infty}(\gamma)$. Let $\gamma'_v = \gamma_v \ominus 1$. We obviously can derive $\Gamma \vdash \mathbb{L}\,[e'] : \tau_{\mathbb{L}} \,/\, \gamma_{\mathbb{L}} \wedge \gamma'_v$ and have $\mathsf{supp}(\gamma'_v) = \mathsf{supp}(\gamma_v)$.

- $\mathbb{L} = \square.X$, similar, with $\gamma'_v = \gamma_v - 1$.

$\square$

Proposition 27 (Nested lift context)
Assume $\Gamma \vdash e : \tau \mathbin{/} \gamma$, $\Gamma \vdash e' : \tau \mathbin{/} \gamma'$, such that $\gamma' = \gamma \wedge \gamma_v$ for some γ_v. If $\Gamma \vdash \mathbb{F}[e] : \tau_{\mathbb{F}} \mathbin{/} \gamma_{\mathbb{F}}$, then there exists γ'_v, with $\mathsf{supp}(\gamma'_v) \subseteq \mathsf{supp}(\gamma_v)$, such that $\Gamma \vdash \mathbb{F}[e'] : \tau_{\mathbb{F}} \mathbin{/} (\gamma_{\mathbb{F}} \wedge \gamma'_v)$. Moreover, $\gamma_{\mathbb{F}} \leq \mathsf{if}_{\text{-}\infty}(\gamma)$.

Preuve By induction on $\mathbb{F}$.

- $\mathbb{F} = \square$. Immediate, with $\gamma'_v = \gamma_v$.
- $\mathbb{F} = \mathbb{L}[\mathbb{F}_1]$. We have a sub-derivation $\Gamma \vdash \mathbb{F}_1[e] : \tau_{\mathbb{L}} \mathbin{/} \gamma_{\mathbb{L}}$. By induction hypothesis, $\gamma_{\mathbb{L}} \leq \mathsf{if}_{\text{-}\infty}(\gamma)$ and there exists γ''_v, with $\mathsf{supp}(\gamma''_v) \subseteq \mathsf{supp}(\gamma_v)$, such that $\Gamma \vdash \mathbb{F}_1[e'] : \tau_{\mathbb{L}} \mathbin{/} (\gamma_{\mathbb{L}} \wedge \gamma''_v)$. By Property 26, $\gamma_{\mathbb{F}} \leq \mathsf{if}_{\text{-}\infty}(\gamma_{\mathbb{L}}) \leq \mathsf{if}_{\text{-}\infty}(\gamma)$, and there exists γ'_v, with $\mathsf{supp}(\gamma'_v) \subseteq \mathsf{supp}(\gamma_v)$, such that $\Gamma \vdash \mathbb{F}[e] : \tau_{\mathbb{F}} \mathbin{/} (\gamma_{\mathbb{F}} \wedge \gamma'_v)$.

$\square$

Proposition 28 (Internal substitution preserves dependencies)
Assume

- $b = (b_v, y \diamond \mathbb{F}[\mathbb{A}[x]], b_1)$,
- $b' = (b_v, y \diamond \mathbb{F}[\mathbb{A}[v]], b_1)$,
- $v = b_v(x)$,
- $\mathsf{dom}(\Gamma_b) = \mathsf{dom}(b)$,
- $\forall (z \diamond_z e_z) \in b, \Gamma + \Gamma_b \vdash e_z : \Gamma_b(z) \mathbin{/} \gamma_z$ *and if* $\diamond_z \neq =_{[?]}$, *then* $\mathsf{TSize}_\circ(\Gamma_b(z)) = \diamond_z$,
- $G = \bigcup_{z \in \mathsf{dom}(b)} (\gamma_z \longrightarrow z)$.

Then, there exist some γ'_z for each $z \in \mathsf{dom}(b)$, such that $\forall (z \diamond_z e'_z) \in b', \Gamma + \Gamma_b \vdash e'_z : \Gamma_b(z) \mathbin{/} \gamma'_z$, with $G' = \bigcup_{z \in \mathsf{dom}(b)} (\gamma'_z \longrightarrow z)$, $G \sqsubseteq G'$, and if $\diamond_z \neq =_{[?]}$, then ${\gamma'_z}^{-1}(0) = \emptyset$.

Preuve As $b_v(x) = v$, we have $\Gamma + \Gamma_b \vdash v : \tau_v \mathbin{/} \gamma_x$. For each $z \in \mathsf{dom}(b) \setminus \{y\}$, take $\gamma'_z = \gamma_z$. By Properties 25 And 27, we obtain that $\gamma_y(x) = \text{-}\infty$ and there exists γ^y_x such that $\mathsf{supp}(\gamma^y_x) \subseteq \mathsf{supp}(\gamma_x)$, and $\Gamma + \Gamma_b \vdash \mathbb{F}[\mathbb{A}[v]] : \Gamma_b(y) \mathbin{/} \gamma_y \wedge \gamma^y_x$. In fact, we let $\gamma'_y = \gamma_y \wedge \text{-}\infty_{|\,\mathsf{supp}(\gamma^y_x)}$. By Lemma 3, we obtain $\Gamma + \Gamma_b \vdash \mathbb{F}[\mathbb{A}[v]] : \Gamma_b(y) \mathbin{/} \gamma'_y$. Moreover, ${\gamma'_y}^{-1}(0) \subseteq {\gamma_y}^{-1}(0) \cup (\text{-}\infty_{|\,\mathsf{supp}(\gamma^y_x)})^{-1}(0) = \emptyset$. Finally, we prove that $G' = G \cup \text{-}\infty_{|\,\mathsf{supp}(\gamma^y_x)} \longrightarrow y$ is less restrictive than G. Now, let $z_1 \xrightarrow{\xi}_{G'} z_2$, with $\xi \neq \infty$. If it is an edge of G, then there is nothing to prove.

Otherwise, it means that $z_2 = y$, $\xi = \text{-}\infty$, and $\gamma^y_x(z_1) = \xi'$. So, $\gamma_x(z_1) \neq \infty$ because $\mathsf{supp}(\gamma^y_x) \subseteq \mathsf{supp}(\gamma_x)$, so $z_1 \xrightarrow{\gamma_x(z_1)}_G x \xrightarrow{\text{-}\infty}_G y$, and therefore $z_1 \xrightarrow{\text{-}\infty}{}^{+}_{G} y$. $\square$

Proposition 29 (Internal substitution)
Assume $e_1 = \texttt{let rec } b_v, y \diamond \mathbb{F}[\mathbb{A}[x]], b \texttt{ in } e_3$, $v = b_v(x)$, and $e_2 = \texttt{let rec } b_v, y \diamond \mathbb{F}[\mathbb{A}[v]], b \texttt{ in } e_3$. If $\Gamma \vdash e_1 : \tau \mathbin{/} \gamma$, then $\Gamma \vdash e_2 : \tau \mathbin{/} \gamma$.

Preuve Let $b_1 = b_v, y \diamond \mathbb{F}[\mathbb{A}[x]], b$ and $b_2 = b_v, y \diamond \mathbb{F}[\mathbb{A}[v]], b$. We have a typing derivation of the shape

$$\frac{\Gamma + \Gamma_{b_1} \vdash e_3 : \tau \mathbin{/} \gamma_{e_3} \qquad \Gamma + \Gamma_{b_1} \vdash b_1 : \Gamma_{b_1} \mathbin{/} G_{b_1} \qquad \vdash_{\lambda_\circ} (G, b_1)}{\Gamma \vdash e_1 : \tau \mathbin{/} \gamma}$$

with

- res fresh,
- $\mathsf{dom}(\Gamma_{b_1}) = \mathsf{dom}(b_1)$,
- $G_{b_1} = \bigcup_{z \in \mathsf{dom}(b_1)} (\gamma_z \longrightarrow z)$,
- $G = G_{b_1} \cup (\gamma_{e_3} \longrightarrow \mathit{res})$,
- and $\gamma \leq (G \gg \mathsf{dom}(b_1) \gg \mathit{res})$.

By Property 28, we have $\forall (z \diamond_z e'_z) \in b', \Gamma + \Gamma_{b_1} \vdash e_z : \Gamma_{b_1}(z) \;/\; \gamma'_z$ with ${\gamma'_z}^{-1}(0) = \emptyset$ and with $G' = \bigcup_{z \in \mathsf{dom}(b)} (\gamma'_z \longrightarrow z)$, $G \sqsubseteq G'$. So, since size indications are unmodified, we conclude immediately with Properties 20 And 5. $\square$

Proposition 30 (External substitution)
Assume $e_1 = \texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}[\mathbb{A}[x]]$, $e_2 = \texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}[\mathbb{A}[v]]$, *with* $v = b_v(x)$. *If* $\Gamma \vdash e_1 : \tau \;/\; \gamma$, *then* $\Gamma \vdash e_2 : \tau \;/\; \gamma$.

Preuve Let $e_3 = \mathbb{F}[\mathbb{A}[x]]$ and $e_4 = \mathbb{F}[\mathbb{A}[v]]$. We have a typing derivation of the shape

$$\frac{\Gamma + \Gamma_{b_v} \vdash e_3 : \tau \;/\; \gamma_{e_3} \qquad \Gamma + \Gamma_{b_v} \vdash b_v : \Gamma_{b_v} \;/\; G_{b_v} \qquad \vdash_{\lambda_\circ} (G, b_v)}{\Gamma \vdash e_1 : \tau \;/\; \gamma}$$

with

- res fresh,
- $\mathsf{dom}(\Gamma_{b_v}) = \mathsf{dom}(b_v)$,
- $G_{b_v} = \bigcup_{z \in \mathsf{dom}(b_v)} (\gamma_z \longrightarrow z)$,
- $G = G_{b_v} \cup (\gamma_{e_3} \longrightarrow \mathit{res})$,
- and $\gamma \leq (G \gg \mathsf{dom}(b_v) \gg \mathit{res})$.

By Properties 25 And 27, we obtain that $\gamma_{e_3}(x) = \text{-}\infty$ (and so $x \xrightarrow{\text{-}\infty}_G \mathit{res}$) and there exists γ'_x such that $\mathsf{supp}(\gamma'_x) \subseteq \mathsf{supp}(\gamma_x)$, $\Gamma + \Gamma_{b_v} \vdash \mathbb{F}[\mathbb{A}[v]] : \tau \;/\; (\gamma_{e_3} \wedge \gamma'_x)$. Let now $\gamma'_{e_3} = \gamma_{e_3} \wedge \gamma'_x$ and $G' = \bigcup_{z \in \mathsf{dom}(b_v)} (\gamma_z \longrightarrow z) \cup (\gamma'_{e_3} \longrightarrow \mathit{res})$. We show that $G \sqsubseteq G'$.

For this, first remark that $G' = G \cup (\gamma'_x \longrightarrow \mathit{res})$. Then, let $z_1 \xrightarrow{\xi}_{G'} z_2$, with $\xi \neq \infty$. If it is an edge of G, then there is nothing to prove.

Otherwise, it means that $z_2 = \mathit{res}$ and $\gamma'_x(z_1) = \xi$. As $\gamma'_x(z_1) \neq \infty$, $\gamma_x(z_1) \neq \infty$ because $\mathsf{supp}(\gamma'_x) \subseteq \mathsf{supp}(\gamma_x)$, so $z_1 \xrightarrow{\gamma_x(z_1)}_G x \xrightarrow{\text{-}\infty}_G \mathit{res}$, and therefore $z_1 \xrightarrow{\text{-}\infty}{}^{+}_G \mathit{res}$.

Thus, we conclude the proof easily. $\square$

Finally, we easily prove the subject reduction property.

Lemma 12 (Subject reduction)
If $e_1 \longrightarrow e_2$ *and* $\Gamma \vdash e_1 : \tau \;/\; \gamma$, *then* $\Gamma \vdash e_2 : \tau \;/\; \gamma$.

Preuve By case analysis on the applied reduction rule, made trivial by the preceding properties. $\square$

B.3 Progress

Now, we prove the standard progress property. A first issue is that when a variable is encountered in a dereferencing context, the reduction replaces it with its definition in the top-level binding, until its intended value is found, that is, a non-variable definition. However, it might never be found, either if the binding contains a cycle of variables (i.e. $x_1 \diamond_1 x_2 \ldots x_{n-1} \diamond_{n-1} x_n$ with $x_1 = x_n$), or if b is incomplete (i.e.$x_1 \diamond_1 x_2 \ldots x_{n-1} \diamond_{n-1} x_n$ with $x_n \notin \mathsf{dom}(b)$). For distinguishing these cases more easily, we define *binding scraping* as follows. It allows us to prove that when well-typed, closed bindings get evaluated, they do not have such defects.

Definition 20 (Scrape bindings)
For any binding b, not necessarily respecting sizes, for any set P of variables, and for any variable $x \in \mathsf{dom}(b)$, we define binding scraping recursively by:

$$\begin{array}{rcll} b^*_P(x) & = & b(x) & \text{if } b(x) \notin \mathsf{Vars} \text{ or } b(x) = y \notin \mathsf{dom}(b) \\ b^*_P(x) & = & \mathtt{cycle} & \text{if } x \in P \text{ and } b(x) \in \mathsf{Vars} \\ b^*_P(x) & = & b^*_{(\{x\} \cup P)}(b(x)) & \text{otherwise.} \end{array}$$

We also define $b^*(x)$ to be $b^*_\emptyset(x)$ if it is a non-variable value, different from $\mathtt{cycle}$.

For the binding $b_1 = (y =_{[n]} x)$, $b_1^*(y)$ is undefined, but ${b_1}^*_\emptyset(y) = x$ is different from $\mathtt{cycle}$. Let us now prove elementary properties of binding scraping.

Proposition 31
Binding scraping is well-defined.

Preuve Let the measure μ be defined from pairs of a binding and a set of variables to natural numbers by $\mu(b, P) = \mid \mathsf{dom}(b) \setminus P \mid$.

First, we notice that if $\mu(b, P) = 0$, then binding scraping immediately returns, on any variable x. Indeed, if $b(x) \notin \mathsf{Vars}$, it returns $b(x)$. Otherwise, if the variable $b(x)$ is in $\mathsf{dom}(b)$, then it is also in P, so $\mu(b, P) = \mathtt{cycle}$, and if the variable $b(x)$ is not in $\mathsf{dom}(b)$, then $\mu(b, P) = b(x)$.

Then, as the measure decreases by 1 at each recursive call, we conclude that $b^*_P(x)$ is well-defined for any $x \in \mathsf{dom}(b)$. $\square$

Proposition 32
For any evaluated b not necessarily respecting sizes, $b^*(x) = b^*_\emptyset(x)$ iff $b^*_\emptyset(x) \notin \mathsf{Vars} \cup \{\mathtt{cycle}\}$.

Proposition 33
If b defines only values $(x \diamond y) \in b$ and $b^*(y)$ is undefined, then either there exists $z \notin \mathsf{dom}(b)$ such that $b^*_\emptyset(y) = z$, or $b^*_\emptyset(y) = \mathtt{cycle}$.

Proposition 34
Assume b defines only values and $\Gamma \vdash \mathtt{let\ rec}\ b, b'\ \mathtt{in}\ e : \tau \ / \ \gamma$, with $\Gamma + \Gamma' \vdash b(x) : \Gamma'(x) \ / \ \gamma_x$ being the immediate sub-derivation corresponding to x. Then, $b^*_\emptyset(x) \neq \mathtt{cycle}$ and there exists γ' such that $\Gamma + \Gamma' \vdash b^*_\emptyset(x) : \Gamma'(x) \ / \ \gamma'$.

Preuve First, we prove that $b^*_\emptyset(x) \neq \mathtt{cycle}$, by contradiction. Assume that b contains bindings of the shape $x_1 \diamond_1 x_2 \ldots x_{n-1} \diamond_{n-1} x_n$, with $x_n = x_1$. Then, for each $i < n$, the degree environment γ_i corresponding to x_i is such that $\gamma_i(x_{i+1}) \leq 0$. But by typing, the underlying dependency cycle is correct, so for all i, $\gamma_i(x_{i+1}) = 0$. However, at least one of these dependency edges $x_i \xrightarrow{0} x_{i+1}$ is backward, since they form a cycle. Let for example $x_{i_0} \xrightarrow{0} x_{i_0+1}$ be backward. By syntactic correctness, $\diamond_{i_0} \neq =_{[?]}$, so there exists a natural number n_0 such that $\diamond_{i_0} = =_{[n_0]}$. But typing also implies that ${\gamma_{i_0}}^{-1}(0) = \emptyset$, which contradicts the fact that the variable y preceding x_{i_0} in the cycle verifies $\gamma_{i_0}(y) = 0$.

Then, we prove the more general property that for any P, if $b^*_P(x) \neq \mathtt{cycle}$, then $\Gamma + \Gamma' \vdash b^*_P(x) : \Gamma'(x) \ / \ \gamma'$ for some γ'. We proceed by induction on the computation of $b^*_P(x)$.

- If $b^*_P(x) = b(x)$, then by typing there is a sub-derivation of $\Gamma + \Gamma' \vdash b^*_P(x) : \Gamma'(x) \ / \ \gamma_x$.

- If $b^*_P(x) = b^*_{\{x\}\cup P}(b(x))$, then $b(x)$ is a variable y in $\mathsf{dom}(b)$. So, by induction hypothesis, as $b^*_{\{x\}\cup P}(y) = b^*_P(x) \neq \texttt{cycle}$, we obtain $\Gamma + \Gamma' \vdash b^*_{\{x\}\cup P}(y) : \Gamma'(y) \;/\; \gamma'$ for some γ'. But, obviously, $\Gamma'(x) = \Gamma'(y)$, which concludes the proof.

□

Finally, we can prove the progress property, stating that a well-typed expression either is a valid answer, or reduces to another expression. Our proof proceeds in two steps. First, we prove that a well-typed expression either is an answer, or reduces to another expression, or needs a variable, i.e. is of the shape $\mathbb{F}[\mathbb{A}[x]]$, or is of the shape $\texttt{let rec}\ b\ \texttt{in}\ e$. Then, we prove the progress property on top level, closed expressions.

Lemma 13 (Partial progress)
If $\Gamma \vdash e : \tau$ and e is not an answer, then either there exists e' such that $e \longrightarrow e'$ or $e = \mathbb{F}[\mathbb{A}[x]]$, or $e = \texttt{let rec}\ b\ \texttt{in}\ f$ for some b and f.

Preuve By induction on e.
If e is not of the shape $\texttt{let rec}\ b\ \texttt{in}\ f$ and is not an answer and is not of the shape $\mathbb{F}[\mathbb{A}[x]]$ for some x, the we distinguish the following cases:

- $e = \mathbb{L}[e_0]$, with $e_0 \notin \mathsf{values}$. If $e_0 = \texttt{let rec}\ b\ \texttt{in}\ f$, then rule Lift applies. Otherwise, by induction hypothesis we are in one of the following cases.
  - $e_0 = \mathbb{F}[\mathbb{A}[x]]$, and e is stuck on x too, i.e. $e = \mathbb{L}[\mathbb{F}[\mathbb{A}[x]]]$.
  - Otherwise, if $e_0 \longrightarrow e_0'$, we reason by case analysis on the applied reduction rule.
    - IM, EM or Subst. This contradicts $e_0 \neq \texttt{let rec}\ b\ \texttt{in}\ f$.
    - Context. Then $e_0 = \mathbb{F}[f]$ and $e_0' = \mathbb{F}[f']$, with $f \rightsquigarrow f'$. Then e reduces by the same rule, since $\mathbb{L}[\mathbb{F}]$ is an evaluation context.
- $e = v_1\ v_2$, and $v_1 \notin \mathsf{Vars}$. By typing, v_1 is a function, and e reduces by rule Beta.
- $e = v.X$, and $v \notin \mathsf{Vars}$. By typing, v is a record defining X, so e reduces by rule Project.

□

Lemma 14 (Progress)
If $\emptyset \vdash e : \tau \;/\; \infty$, then either e is an answer, or there exists e' such that $e \longrightarrow e'$.

Preuve By Lemma 13, if e is not an answer, there are only three possibilities. If e reduces to some e', then there is nothing to prove. Otherwise, if $e = \mathbb{F}[\mathbb{A}[x]]$, it contradicts the well-typedness of e. Otherwise, $e = \texttt{let rec}\ b\ \texttt{in}\ f$. First, if f has the shape $\texttt{let rec}\ b'\ \texttt{in}\ g$, then rule EM applies. Otherwise, we proceed by case on b.

1. If b defines only values, then we distinguish two cases. If b does not respect sizes, then let b_v be its maximal prefix respecting sizes. The binding b has a prefix of the shape $b_v, x =_{[n]} v$, with v not of size n. If v is a variable y, then by typing $y \in \mathsf{dom}(b)$ and the grah G of b contains an edge $y \xrightarrow{-\infty}_G x$, so by ordered correctness, $y \in \mathsf{dom}(b_v)$, and e reduces by rule Subst.

   Otherwise, $b = b_v$. f cannot be an answer, because otherwise, either it would have the shape $\texttt{let rec}\ {b_v}'\ \texttt{in}\ v$ which contradicts $f \neq \texttt{let rec}\ b'\ \texttt{in}\ g$, or it would be a value, and so e would be an answer itself.

   So, by Lemma 13, we are in one of the two following cases.

   - $f \longrightarrow f'$. By case analysis on the reduction:
     - IM, EM, or Subst. Contradicts $f \neq \texttt{let rec}\ b'\ \texttt{in}\ g$.

- – CONTEXT. We have $f = \mathbb{F}[g]$ and $f' = \mathbb{F}[g']$, with $g \rightsquigarrow g'$. So, rule CONTEXT applies for e as well, since $\texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}$ is an evaluation context.
- $f = \mathbb{F}[\mathbb{A}[x]]$. Then, $e = \texttt{let rec}\ b_v\ \texttt{in}\ \mathbb{F}[\mathbb{A}[x]]$. By typing, we have $x \in \mathsf{dom}(b_v)$, so rule SUBST applies.

2. If b does not define only values, then b is of the shape $b_0, y \diamond g, b_1$, where b_0 defines only values and g is not a value. As above, if b_0 does not respect sizes, then e reduces by rule SUBST. Otherwise $b_0 = b_v$. If g is of the shape $\texttt{let rec}\ b'\ \texttt{in}\ g'$, then rule IM applies. Otherwise, g cannot be a result, so we are in one of the following cases, by Lemma 13.

   - If $g \longrightarrow g'$, by case on the reduction.
     - IM, EM, or SUBST. Contradicts $g \neq \texttt{let rec}\ b'\ \texttt{in}\ g'$.
     - CONTEXT : then $g = \mathbb{F}[g_0]$ and $g' = \mathbb{F}[g_0']$, with $g_0 \rightsquigarrow g_0'$, so the global context is an evaluation context and rule CONTEXT applies for e.
   - If $g = \mathbb{F}[\mathbb{A}[x]]$. By case on $\mathbb{F}$. First, we know that $x \notin \mathsf{dom}(y = g, b_1)$, since by Property 27, x has degree $-\infty$ in the degree environment for y. Moreover, by typing $x \in \mathsf{dom}(b_v)$, so rule SUBST applies.

□

Finally, we obtain a standard soundness theorem.

Theorem 2 (Soundness)
The evaluation of a closed well-typed expression may either not terminate or reach an answer.

References

[1] Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):95–178, 2002.

[2] Gérard Boudol. The recursive record semantics of objects revisited. Research report 4199, INRIA, 2001. Preliminary version presented at ESOP'01, LNCS 2028.

[3] Gérard Boudol and Pascal Zimmer. Recursion in the call-by-value lambda-calculus. Fixed Points in Computer Science, 2002.

[4] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[5] Derek R. Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *30th symposium Principles of Programming Languages*, 2003.

[6] Derek R. Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, Pittsburgh, PA, March 2001.

[7] Derek R. Dreyer, Robert Harper, and Karl Crary. A type system for well-founded recursion. Technical Report CMU-CS-03-163, Carnegie Mellon University, 2003.

[8] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[9] Tom Hirschowitz. *Mixin modules, modules, and extended recursion in a call-by-value setting.* PhD thesis, University of Paris VII, December 2003.

[10] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In Daniel Le Métayer, editor, *European Symposium on Programming*, volume 2305 of *LNCS*, pages 6–20, 2002.

[11] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. Submitted for publication to HOSC, 2004.

[12] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. *Higher-Order and Symbolic Computation*, 2005. to appear.

[13] Richard Kelsey, William Clinger, and Johnathan Rees. The revised^5 report on the algorithmic language scheme, 1998.

[14] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[15] Xavier Leroy. A proposal for recursive modules in Objective Caml. Available on the Web, `http://pauillac.inria.fr/~xleroy/publi/recursive-modules-note.pdf`, 2003.

[16] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml 3.06 reference manual*, 2002. Available on the Web, `http://caml.inria.fr/`.

[17] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2003.

[18] David B. MacQueen. Modules for Standard ML. *Lisp and Functional Programming*, pages 198–207, 1984.

[19] Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. The MIT Press, 1990.

[20] Chris Okasaki. *Purely Functional Data Structures*, chapter 10. Cambridge University Press, 1998.

[21] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1992.