A reduction semantics for call-by-value mixin modules

Tom Hirschowitz, Xavier Leroy, Joe B. Wells

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A reduction semantics for call-by-value mixin modules

Tom Hirschowitz — Xavier Leroy — Joe B. Wells

N° 4682

Décembre 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4682--FR+ENG

A reduction semantics for call-by-value mixin modules

Tom Hirschowitz * , Xavier Leroy * , Joe B. Wells †

Thème 2 — Génie logiciel
et calcul symbolique
Projet Cristal

Rapport de recherche n° 4682 — Décembre 2002 — 34 pages

Abstract: Module systems are important for software engineering: they facilitate code reuse without compromising the correctness of programs. However, they still lack some flexibility: first, they do not allow mutually recursive definitions to span module boundaries ; second, definitions inside modules are bound early, and cannot be overridden later, as opposed to inheritance and overriding in class-based object-oriented languages, which follow the late binding semantics. This paper examines an alternative, hybrid idea of modularization concept, called *mixin modules*. We develop a language of call-by-value mixin modules with a reduction semantics, and a sound type system for it, guaranteeing that programs will run correctly.

Key-words: modularity, mixin modules, recursion, type systems, reduction semantics

* Inria Rocquencourt, projet Cristal
† Heriot-Watt University, Edinburgh, UK

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Sémantique à réduction pour un langage de modules mixins en appel par valeur

Résumé : Les sytèmes de modules jouent un rôle important dans l'ingénierie logicielle : ils facilitent le partage de code, tout en conservant de bonnes propriétés de sûreté d'exécution. Leur flexibilité reste cependant limitée. D'une part, ils interdisent les références mutuelles entre modules. D'autre part, une définition dans un module est liée statiquement et ne peut donc pas être redéfinie, au contraire des méthodes dans les langages à objets avec classes, qui sont en liaison tardive. Ce travail examine une idée à mi-chemin entre les classes et les modules, appelée *modules mixins*. On présente un langage de modules mixins en appel par valeur, doté d'une sémantique opérationnelle à réduction et d'un système de typage garantissant l'exécution correcte des programmes.

Mots-clés : modularité, modules mixins, récursion, typage, sémantique à réduction

1 Introduction

Code reuse is an important aspect of software engineering. It can take place at several levels. Linguistic abstractions, such as higher-order functions, classes and objects, increase code reuse inside a program. Separate compilation provides for libraries, which allow programs written in the same language to share code. Components allow code reuse to spread across different languages and different physical sites.

Besides their expressiveness, one has to consider these concepts with respect to program safety and correctness. Language abstractions have been extensively studied from this standpoint, specifically through the use of sound type systems: there are well-known ways for ensuring statically (i.e. at compile time) that an object-oriented or a functional program will not go wrong (see e.g. [22]). Separate compilation has been investigated, and sound type systems have been elaborated, which are able to statically prove that a separately compiled program will not go wrong [15, 12, 7, 19]. The component-based approach is in its early stage of formalization [20].

Among linguistic constructs, module systems are of particular interest, because they allow dealing with separate compilation at the language level [15, 12]. One the one hand, modules are language constructs that can be described with mostly standard formalisms, such as operational semantics and type systems. One the other hand, they can also be viewed as compilation units, and thus express strategies for separate compilation and linking [15, 7, 19].

A well-known module system is the one of ML [17, 16] support type abstraction, parameterization over other modules, and separate compilation. However, it still lacks some flexibility: first, it does not allow mutually recursive definitions to span module boundaries ; second, definitions inside modules are bound early, and cannot be overridden later, as opposed to inheritance and overriding in class-based object-oriented languages, which follow the late binding semantics.

This paper examines an alternative, hybrid idea of modularization concept, called *mixin modules*. The original idea appeared in the early 90's with Bracha, Cook, and Lindstrom [5, 4, 6], and was further developed by Duggan and Sourelis [8], Flatt and Felleisen [11, 10], Ancona and Zucca [1, 2], Wells and Vestergaard [21], and Hirschowitz and Leroy [14]. It consists of a module language – a modularization construct independent from the core language – with features for incremental programming, inspired by *mixin classes*. Basically, a mixin module is a collection of named definitions and declarations. Declarations may be filled with definitions by composition with another mixin module. The definitions of one mixin module then fill the corresponding declarations of the other one, according to their names. Definitions are not statically bound to one another, and may be redefined.

The present work tries to address certain limitations of earlier proposals for mixin modules. Bracha et al. allow only values to be defined in mixin modules, which is too restrictive. Flatt and Felleisen and Duggan and Sourelis introduce a notion of initialization section for extending the expressive power of mixin modules, but they lose the ability to modify a mixin module *a posteriori*. Ancona and Zucca and Wells and Vestergaard allow any expression to be defined in a mixin module, keeping the ability to modify a mixin module *a posteriori*. However, their proposals only work in call-by-name, and are incompatible with call-by-value programming languages. Hirschowitz and Leroy develop a call-by-value calculus of mixin modules. However, the semantics of their calculus is not given by reduction rules as usual, but by a complicated type-directed translation to a non-standard λ-calculus. This makes reasoning on programs almost impossible.

The present paper gives a reduction semantics for call-by-value mixin modules in the style of [14], supporting all the mixin operators proposed by Bracha [4]. Section 3 presents the calculus, named **MM**, and its semantics. A type system is defined in section 4, and it is proved sound. Conclusions and future work are reviewed in 5.

2 Overview

2.1 Mixin modules

A mixin module is an unordered, unevaluated, possibly incomplete module: it is a set of named definitions and declarations.

Consider the following mixin module, in an OCaml-like syntax:

```
mixin A =
  import
    val x : int
    val f : int -> int
```

```
  export
    define y = (g 0) + x
    define g z = ... f ...
  end
```

The declaration `val x : int` is used by the definition `define y = (g 0) + x`.
The declaration `val f : int -> int` is used by the definition `define g z = ... f ...`
The scope is mutually recursive, as illustrated by the definition `define y = (g 0) + x`, depending on `g`.

The operator for linking mixin modules is composition `+`, which combines two mixin modules, filling the declarations of one argument with the definitions of the other, and *vice versa*. Consider the following mixin module.

```
mixin B =
  import
    val y : int
    val g : int -> int
  export
    define x = y + 1
    define f z = ... g ...
  end
```

The composition `mixin C = A + B` of `A` and `B` is equivalent to the mixin module:

```
mixin C =
  import
  export
    define y = (g 0) + x
    define g z = ... f ...
    define f z = ... g ...
    define x = y + 1
  end
```

The declarations of one mixin module are replaced with the similarly named definitions of the other. The export section is the concatenation of the export sections of `A` and `B`. The code remains unevaluated, so the evaluation of `C` does not go wrong. However, there is an ill-founded recursion between `x` and `y`, and if we try to evaluate the code contained by `C`, a dynamic error will occur. Fortunately, mixin modules feature late binding: one may delete the definition of `x` in `B`, thanks to the delete operator `|-`.

```
mixin B' =
  import
    val x : int
    val y : int
    val g : int -> int
  export
    define f z = ... g ...
  end
```

A new definition for `x` may be defined in another mixin module:

```
mixin D = import
          export
            define x = 0
          end
```

The mixin module `E = A + B' + D` is equivalent to

```
mixin E = import
          export
            define y = (g 0) + x
            define g z = ... f ...
```

```
    define f z = ... g ...
    define x = 0
  end
```

Now, all holes are filled, and the mixin module can be instantiated. It is the role of the `close` operator, which generates a module out of a mixin module without holes: `module M = close E`. The semantics of `close` includes a reordering of definitions, in order to avoid references to a not yet evaluated definition. The initial ordering is kept, as far as possible. Here, it results in only moving the definition of `y`, because it needs the values of `g` and `x` (and possibly `f`) to evaluate. The definition `module M = close E` is equivalent to:

```
module M = struct
  let rec g z = ... f ...
          f z = ... g ...
  let x = 0
  let y = (g 0) + x
end
```

The evaluation of `M` consists in successively evaluating the definitions, and returning the evaluated module:

```
module M = struct
  let rec f z =  ... g ...
      and g z = ... f ...
  let x = 0
  let y = V
end
```

(Where `V` is the result of `(g 0) + x`.)

We refer to [4, 2] for more details on mixin modules and other operators.

2.2 An extended binding construct

In **MM**, the definitions of `x` and `y` could not have been included in the mutually recursive definition of `f` and `g`. Indeed, the `let rec` construct of ML only allows to bind syntactic functions (or constructed values in the case of OCaml). Therefore, in the case of more complex dependencies between the definitions of a mixin module, instantiation would lead to nested `let` and `let rec` bindings. In order to avoid this complication, our calculus features a slightly more powerful `let rec` than that of ML, which is reminiscent of monadic recursive bindings [9]. It evaluates the definitions from left to right, and basically only goes wrong when the value of a variable defined to the right of the current definition is needed. For instance, the definition

```
let rec f x = ... g ...
        g x = ... f ...
        x = 0
        y = (g 0) + x
```

evaluates correctly: `f`, `g`, and `x` are already values, and `y` is defined last.

Notice that the body of `f` makes a reference to `g`, which is defined to the right of it. We call such a reference a forward reference. A forward reference is syntactically correct if it points to an expression of predictable shape. In the above example, the definition of `g` is a syntactic abstraction, which is considered an expression of predictable shape. A forward reference is semantically correct if it does not require the value of the referenced variable. In the above example, the definition of `g` is already evaluated, so it doesn't need to inspect the value of `f`.

2.3 Typing issues

Our `let rec` is not much more powerful than that of ML. Its main interest is that complex series of sequential `let` bindings and mutually recursive `let rec` bindings are now written as straightforward definitions. Its typing is much less straightforward of course, since it requires the analysis of dependencies between the definitions. This analysis has to go beyond immediate dependencies, as shown by the following example.

Example 1 *Consider the following binding, where braces enclose records and* `X` *and* `Z` *are record field names.*

	x	$\in$ **Vars**	Variable
	X	$\in$ **Names**	Name
Expression:	e	$::= x$	Variable
		$\mid \{X_1 = e_1 \dots X_n = e_n\}$	Record
		$\mid e.X$	Record selection
		$\mid$ **let rec** $x_1 = e_1 \dots x_n = e_n$ **in** e	**let rec**
		$\mid \langle X_1 \triangleright x_1 \dots X_n \triangleright x_n \mid d_1 \dots d_m \rangle$	Structure
		$\mid e_1 + e_2 \mid \nabla e \mid e \,!\, X$	Composition, closure, freezing
		$\mid e_{\mid X_1 \dots X_n} \mid e_{\mid -X_1 \dots X_n}$	Projection, deletion
		$\mid e_{:X_1 \dots X_n} \mid e_{:-X_1 \dots X_n}$	Showing, hiding
		$\mid e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$	Renaming
		$\mid e_{X \succ Y}$	Splitting
Definition:	d	$::= X[x_1 \dots x_n] \triangleright x = e$	Named definition
		$\mid \_[x_1 \dots x_n] \triangleright x = e$	Anonymous definition

Figure 1: Syntax of **MM**

```
let rec x = { X = z }
        y = x.X.Z
        z = { Z = 0 }
```

There is a forward reference from `x` *to* `z`*, but the definition of* `z` *is of predictable shape, so the expression is syntactically correct. Moreover, there are no forward references needing the value of the referenced definition. One could expect it to be a sufficient condition for the binding not to go wrong because of dependencies. Unfortunately, the evaluation of the definition of* `y` *needs both the values of* `x` *and* `z`*.*

Roughly, the correct requirement is that no forward reference path starts with a strict dependency. We say that a definition `x = M` strictly depends on another one `y = N`, when the evaluation of `M` might require the value of `y`. What does "might require" mean here? It is a very restrictive syntactic approximation: the only case where we detect that an expression `M` will not need the value of one of its free variables `x` is when `M` is a value of predictable shape. In example 1, there is a forward reference path from `x` to `z`, which does not end with a strict dependency, since `{ X = z }` is a value of predictable shape. However, this path extends to a forward reference path from `y` to `z`, which starts with a strict dependency. Therefore, the binding is rejected by the type system.

We have seen that mixin modules are instantiated by the `close` operator, which generates a binding out of them. In order to statically ensure that this binding is correct, the type system keeps track of the dependencies between mixin components. The type of a mixin contains both type information about its components, and a graph representing their dependencies. When composing two mixin modules, the type system takes the union of their dependency graphs. When a concrete mixin (a mixin with no declarations, only definitions), gets instantiated, its graph is required not to have cycles with strict dependencies. This is sufficient: if there is no cycle with strict dependencies, then an ordering of definitions can be found, such that no forward reference path starts with a strict dependency. The `close` operator finds this ordering.

3 Definition of the MM language

3.1 Syntax

The syntax of **MM** is defined in figure 1. The meta-variables X and x range over names and variables, respectively. Variables are used as binders, as usual. Names are used for accessing to definitions in mixin modules, as an external interface to other parts of the expression. Figure 2 recapitulates the meta-variables and notations we introduce in the remainder of this section.

s	$::= X_1 = e_1 \dots X_n = e_n$	Record
b	$::= x_1 = e_1 \dots x_n = e_n$	Binding
ι	$::= X_1 \triangleright x_1 \dots X_n \triangleright x_n$	Input (injective)
o	$::= d_1 \dots d_2$	Output
r	$::= X_1 \mapsto Y_1 \dots X_n \mapsto Y_n$	Renaming (injective)
$op[e]$	$::= e.X$	Record selection
	$\mid \nabla e \mid e \,!\, X$	Closure, freezing
	$\mid e_{\mid X_1 \dots X_n} \mid e_{\mid -X_1 \dots X_n}$	Projection, deletion
	$\mid e_{:X_1 \dots X_n} \mid e_{:-X_1 \dots X_n}$	Showing, hiding
	$\mid e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$	Renaming
	$\mid e_{X \succ Y}$	Splitting

For a finite map f, and a set of variables P,
$\mathbf{dom}(f)$ is its domain, $\mathbf{cod}(f)$ is its codomain
$f_{\mid P}$ is its restriction to P, and $f_{\setminus P}$ is its restriction to $\mathbf{Vars} \setminus P$.

Figure 2: Meta-variables and notations

Expressions include variables x, records (labeled by names) $\{X_1 = e_1 \dots X_n = e_n\}$, and record selection $e.X$, which are standard.

MM features mutually recursive bindings of the shape **let rec** b **in** e (where b is a list of definitions $x_1 = e_1 \dots x_n = e_n$). Note that there is no restriction to binding only value forms.

Expressions also include structures. A structure is a pair of an input ι of the shape $X_1 \triangleright x_1 \dots X_n \triangleright x_n$, and of an output o of the shape $d_1 \dots d_m$. ι maps external names imported by the structure to internal variables (used in o). o is a list (the order matters) of definitions d. A definition is of the shape $L[x_1 \dots x_n] \triangleright x = e$, where the label L may be either a name X or the anonymous label _ and e is the body of the definition. The possibly empty finite set of names $x_1 \dots x_n$ is the set of fake dependencies of this definition on other definitions of the structure. (This allows the programmer to force an order of evaluation.)

Finally, **MM** follows the literature about mixin modules [4, 2, 14] in its set of operators, including composition $e_1 + e_2$, closure ∇e, freezing $e \,!\, X$, projection $e_{\mid X_1 \dots X_n}$, deletion $e_{\mid -X_1 \dots X_n}$, showing $e_{:X_1 \dots X_n}$, hiding $e_{:-X_1 \dots X_n}$, and renaming $e[X_1 \mapsto Y_1 \dots X_n \mapsto Y_n]$. There is a new operator called splitting $e_{X \succ Y}$. We let op range over the set of operators (see figure 2), and denote by $op[e]$ the application of op to the expression e.

Syntactic correctness Renamings $r = (X_1 \mapsto Y_1 \dots X_n \mapsto Y_n)$, inputs $\iota = (X_1 \triangleright x_1 \dots X_n \triangleright x_n)$, records $s = (X_1 = e_1 \dots X_n = e_n)$, bindings $b = (x_1 = e_1 \dots x_n = e_n)$, are required to be finite maps: a renaming is a finite map from names to names, an input is a finite map from names to variables, a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice. Renamings and inputs are required to be injective. Outputs $o = (d_1 \dots d_n)$ are required not to define the same name twice, and not to define the same variable twice. Structures are required not to define the same name twice and not to define the same variable twice. Fake dependencies in a definition must be bound in the same structure.

In a **let rec** binding $b = (x_1 = e_1 \dots x_n = e_n)$, when for some $1 \leq i \leq j \leq n$, $x_j \in \mathbf{FV}(e_i)$, we say that there is a forward reference from x_i to x_j. Forward references in bindings are syntactically forbidden, except when they point to a certain class of expressions, the class of expressions with a predictable shape. We approximate that the shape of an expression is predictable if it is a structure, a record, or a binding followed by an expression of predictable shape. Formally $e_\downarrow \in \mathbf{Predictable} ::= \{o\} \mid \langle \iota \mid o \rangle \mid \mathbf{let\ rec}\ b\ \mathbf{in}\ e_\downarrow$.

Sequences Outputs may be viewed as finite maps from pairs of a label and a variable (L, x) to pairs of a finite set of variables $(x_1 \dots x_n)$ and an expression e. Renamings, inputs, records, bindings, and outputs are often considered as finite maps in the sequel. We refer to them collectively as sequences, and use the usual notions on finite maps, such as the domain **dom**, the codomain **cod**, the restriction $\cdot_{\mid P}$ to a set P, or the co-restriction $\cdot_{\setminus P}$ outside of a set P. Notice that the codomain of an output o, restricted to pairs of a name and a variable (no anonymous label), may in turn be viewed as an input, since it is an injective finite map. We denote it by $\mathbf{Input}(o)$.

Value:	$v ::= x \mid \{s_v\}$
	$\mid \langle X_1 \triangleright x_1 \dots X_n \triangleright x_n \mid d_1 \dots d_n \rangle$
Answer:	$res ::= v \mid \textbf{let rec } b_v \textbf{ in } v$
Value sequence:	$s_v ::= X_1 = v_1 \dots X_1 = v_1$
	$b_v ::= x_1 = v_1 \dots x_n = v_n$

Figure 3: Values in **MM**

$$\langle \iota_1 \mid o_1 \rangle \square \langle \iota_2 \mid o_2 \rangle \text{ means } \begin{cases} \langle \iota_1 \mid o_1 \rangle \sqsubset \langle \iota_2 \mid o_2 \rangle \text{ and} \\ \langle \iota_2 \mid o_2 \rangle \sqsubset \langle \iota_1 \mid o_1 \rangle. \end{cases}$$

$\langle \iota_1 \mid o_1 \rangle \sqsubset \langle \iota_2 \mid o_2 \rangle$ means that for all $(L \triangleright x) \in \textbf{dom}(\langle \iota_1 \mid o_1 \rangle)$,
$x \in \textbf{FV}(o_2) \cup \textbf{Variables}(\langle \iota_2 \mid o_2 \rangle) \Rightarrow (L \triangleright x) \in \textbf{dom}(\langle \iota_2 \mid o_2 \rangle)$ and $L \in \textbf{Names}$.

Figure 4: Definition of $\square$

Structural equivalence We consider the expressions equivalent up to alpha-conversion of binding variables in structures and **let rec** expressions. In the following, we assume that no undue variable capture occurs.

3.2 Semantics

The semantics of **MM** is defined in two steps: a contraction relation describes the action of the operators, and a reduction relation extends it properly to any expression.

Values As defined in figure 3, an **MM** value is either a variable x, or an evaluated record $\{X_1 = v_1 \dots X_1 = v_1\}$, or a structure $\langle \iota \mid o \rangle$. A valid result of the evaluation of an **MM** expression is a value, possibly surrounded by an evaluated binding. It thus has the shape **let rec** $x_1 = v_1 \dots x_n = v_n$ **in** v. The meta-variables s_v and b_v respectively range over evaluated record sequences and bindings.

The contraction relation The contraction relation is defined by the rules in figure 5, where for any sets $P_1 \dots P_n$, $P_1 \perp \dots \perp P_n$ means that the P_i's are pairwise disjoint.

The first rule (LIFT) describes how **let rec** bindings are lifted up to the top of the term. When the evaluation of a sub-expression results in a **let rec** binding, **MM** lifts it one level up, as follows. Lift contexts $\mathbb{L}$ are defined as

$$\begin{array}{rcl} \mathbb{L} & ::= & \{\mathbb{S}\} \mid op[\square] \mid \square + e \mid v + \square \\ \mathbb{S} & ::= & s_v, X = \square, s. \end{array}$$

Rule (LIFT) states that an expression of the shape $\mathbb{L}[\textbf{let rec } b \textbf{ in } e]$ evaluates to
let rec b **in** $\mathbb{L}[e]$, provided no variable capture occurs.

The record selection rule (SELECT) straightforwardly describes the selection of a record field.

The rules for mixin deletion (DELETE) and projection (PROJECT) are dual. Rule (DELETE) describes how **MM** deletes a finite set of names $X_1 \dots X_n$ from a structure $\langle \iota \mid o \rangle$. First, o is restricted to the other definitions, to obtain $o_{\setminus \{X_1 \dots X_n\}}$ (which is shorthand for $o_{\setminus \{X_1 \dots X_n\} \times \textbf{Vars}}$). Second, the removed definitions remain bound as inputs, by adding the corresponding inputs to ι.

Rule (PROJECT) describes how **MM** projects a mixin to some finite set of names $X_1 \dots X_n$ from a structure $\langle \iota \mid o \rangle$. First, o is restricted to the corresponding definitions and to the local ones, to obtain $o_{\mid \{\_, X_1 \dots X_n\}}$ (which is a shorthand for $o_{\mid \{\_, X_1 \dots X_n\} \times \textbf{Vars}}$). Then, the removed definitions remain bound as inputs, by adding the corresponding inputs to ι.

Rules (SHOW) and (HIDE) are dual. Rule (SHOW) allows to hide all the exported names of a structure, except the given ones. It proceeds by making the other definitions local, as defined by

$$\textbf{Show}(L[y^*] \triangleright x = e, \mathcal{N}) = \begin{cases} L[y^*] \triangleright x = e \text{ if } L \in \mathcal{N} \\ \_[y^*] \triangleright x = e \text{ otherwise.} \end{cases}$$

$$\frac{\mathbf{dom}(b) \perp \mathbf{FV}(\mathbb{L})}{\mathbb{L}[\mathbf{let\ rec}\ b\ \mathbf{in}\ e] \rightsquigarrow_c \mathbf{let\ rec}\ b\ \mathbf{in}\ \mathbb{L}[e]} \quad (\textsc{Lift}) \qquad \{X_1 = v_1 \ldots X_n = v_n\}.X_i \rightsquigarrow_c v_i \quad (\textsc{Select})$$

$$\langle \iota \mid o \rangle_{|-X_1 \ldots X_n} \rightsquigarrow_c \langle \iota, \mathbf{Input}(o)_{|\{X_1 \ldots X_n\}} \mid o_{\setminus\{X_1 \ldots X_n\}} \rangle \quad (\textsc{Delete})$$

$$\langle \iota \mid o \rangle_{|X_1 \ldots X_n} \rightsquigarrow_c \langle \iota, \mathbf{Input}(o)_{\setminus\{X_1 \ldots X_n\}} \mid o_{|\{\_, X_1 \ldots X_n\}} \rangle \quad (\textsc{Project})$$

$$\langle \iota \mid o \rangle_{:X_1 \ldots X_n} \rightsquigarrow_c \langle \iota \mid \mathbf{Show}(o, \{X_1 \ldots X_n\}) \rangle \quad (\textsc{Show})$$

$$\langle \iota \mid o \rangle_{:-X_1 \ldots X_n} \rightsquigarrow_c \langle \iota \mid \mathbf{Show}(o, \mathbf{Names} \setminus \{X_1 \ldots X_n\}) \rangle \quad (\textsc{Hide})$$

$$\langle \iota \mid o_1, X[y^*] \triangleright x = e, o_2 \rangle \,!\, X \rightsquigarrow_c \langle \iota \mid o_1, \_[y^*] \triangleright x = e, o_2, X \triangleright \_ = x \rangle \quad (\textsc{Freeze})$$

$$\frac{\mathbf{Names}(\langle \iota \mid o \rangle) \perp (\mathbf{cod}(r) \setminus \mathbf{dom}(r))}{\langle \iota \mid o \rangle[r] \rightsquigarrow_c \langle \iota\{r\} \mid o\{r\} \rangle} \quad (\textsc{Rename})$$

$$\langle \iota \mid o_1, X[z^*] \triangleright x = e, o_2 \rangle_{X \succ Y} \rightsquigarrow_c \langle \iota, X \triangleright x \mid o_1, Y[z^*] \triangleright \_ = e, o_2 \rangle \quad (\textsc{Split})$$

$$\frac{\langle \iota_1 \mid o_1 \rangle \,\square\, \langle \iota_2 \mid o_2 \rangle \qquad \mathbf{Names}(o_1) \perp \mathbf{Names}(o_2)}{\langle \iota_1 \mid o_1 \rangle + \langle \iota_2 \mid o_2 \rangle \rightsquigarrow_c \langle (\iota_1 \cup \iota_2) \setminus \mathbf{Input}(o_1, o_2) \mid o_1, o_2 \rangle} \quad (\textsc{Sum})$$

$$\frac{\mathbf{Bind}(\overline{o}) \text{ is correct}}{\nabla\langle \emptyset \mid o \rangle \rightsquigarrow_c \mathbf{let\ rec}\, \mathbf{Bind}(\overline{o})\, \mathbf{in}\, \mathbf{Record}(\overline{o})} \quad (\textsc{Close})$$

Figure 5: Computational contraction relation

Evaluation context:
$\mathbb{E} ::= \mathbb{F} \mid \textbf{let rec } b_v \textbf{ in } \mathbb{F} \mid \textbf{let rec } \mathbb{B}[\mathbb{F}] \textbf{ in } e$
Lift context:
$\mathbb{L} ::= \{\mathbb{S}\} \mid op[\Box] \mid \Box + e \mid v + \Box$

Multiple lift context:
$\mathbb{F} ::= \Box \mid \mathbb{L}[\mathbb{F}]$
Binding context:
$\mathbb{B} ::= b_v, x = \Box, b$
Record context:
$\mathbb{S} ::= s_v, X = \Box, s$

Figure 6: Evaluation contexts

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

Figure 7: Access in evaluation contexts

Rule (Hide) symmetrically hides the given names in a structure. It proceeds by showing the other ones.

Rule (Freeze) describes how a name X is frozen in a structure $\langle \iota \mid o \rangle$. First, the corresponding definition $X[y^*] \triangleright x = e$ is made local, by replacing X with the local label _. Then, a new definition is added at the end of the output. It is named X, is bound to a fresh variable (denoted by _ in the rule by abuse of notation), and is defined by referring to x.

Renaming of a structure $\langle \iota \mid o \rangle$ by a renaming r, defined by rule (Rename), replaces the names in ι and o with the new ones. Formally, for $\mathcal{N} \subset \textbf{Names}$, we define $r_{\mathcal{N}}$ by $r \cup id_{|\mathcal{N} \setminus \textbf{dom}(r)}$ and for a finite map f with $\textbf{dom}(f) \subset \textbf{Names}$, we define $f\{r\}$ by $f \circ (r_{\textbf{dom}(f)})^{-1}$. The finite map $f\{r\}$ is well-defined provided $r_{\textbf{dom}(f)}$ is injective, which holds as soon as $\textbf{cod}(r) \cap \textbf{dom}(f) \subset \textbf{dom}(r)$ or in other words $\textbf{dom}(f) \perp (\textbf{cod}(r) \setminus \textbf{dom}(r))$. By the side-condition $\textbf{Names}(\langle \iota \mid o \rangle) \perp (\textbf{cod}(r) \setminus \textbf{dom}(r))$, this is the case for $\iota\{r\}$. (We denote by $\textbf{Names}(\langle \iota \mid o \rangle)$ the set of names bound by the structure, i.e. $\textbf{dom}(\iota) \cup \textbf{dom}(\textbf{Input}(o))$.) Finally, we define $o\{r\}$ by $o \circ (r_{\textbf{Names}(o)}, id_{\textbf{Vars}})^{-1}$, with the order kept from o, and where $(f_1, f_2)(x_1, x_2) = (f_1(x_1), f_2(x_2))$. Notice that when composing two functions $f \circ g$, we consider a function whose domain is $g^{-1}(\textbf{dom}(f))$ and on this domain is $f(g(x))$. In the rule, $o\{r\}$ is well-defined, thanks to the side-condition. Syntactic correctness is preserved, since $r_{\textbf{Names}(\langle \iota \mid o \rangle)}$ is injective. So, after renaming, no name is defined twice.

The (Split) rule introduces a new operator "split". If there is a definition $X[z^*] \triangleright x = e$ for the name X in $\langle \iota \mid o \rangle$, the split operator $\langle \iota \mid o \rangle_{X \succ Y}$ splits it into an input $X \triangleright x$ and a definition $Y[z^*] \triangleright y = e$ (with a fresh y). References to x continue referencing it as an input, but the former definition e remains exported as Y. The operation is different from renaming X to Y or deleting X.

The (Sum) rule defines the composition of two structures $\langle \iota_1 \mid o_1 \rangle$ and $\langle \iota_2 \mid o_2 \rangle$. The result is a structure $\langle \iota \mid o \rangle$, defined as follows. ι is the union of ι_1 and ι_2, where names defined in o_1 or o_2 are removed. o is defined as the concatenation of o_1 and o_2. The side condition $\langle \iota_1 \mid o_1 \rangle \square \langle \iota_2 \mid o_2 \rangle$ checks that both structures agree on bound variables, and that no free variable is captured. It is defined in figure 4, where $\textbf{dom}(\langle \iota \mid o \rangle) = \iota \cup \textbf{dom}(o)$, and $\textbf{Variables}(\langle \iota \mid o \rangle)$ denotes $\textbf{cod}(\iota) \cup \{x \mid (L, x) \in \textbf{dom}(o)\}$. Lastly, o_1 and o_2 are required not to define the same names.

Eventually, the (Close) rule describes the instantiation of a structure $\langle \iota \mid o \rangle$. ι must be empty. The instantiation is in three steps. First, o is reordered to $\overline{o}$, according to its dependencies, to its fake dependencies, and to its default ordering. Second, a binding $\textbf{Bind}(\overline{o})$ is generated, defining, for each definition $d = L[y^*] \triangleright x = e$ in o, the definition $x = e$, in the same order as in $\overline{o}$. Third, the named definitions of $\overline{o}$ are put in a record $\textbf{Record}(\overline{o})$, with, for each named definition $X[y^*] \triangleright x = e$, a field $X = x$, and this record is the result of the instantiation. The side condition ensures that the generated binding is syntactically correct, especially that there is no forward reference to bindings of unpredictable shapes.

The reduction relation The reduction relation is defined by the rules in figure 8, using notions defined in figures 6 and 7.

Rule (Context) extends the contraction relation to any evaluation context. Evaluation contexts are defined in figure 6. We call a multiple lift context $\mathbb{F}$ a series of nested lift contexts. An evaluation context $\mathbb{E}$ is a multiple lift context, possibly inside a partially evaluated binding, or under a fully evaluated binding. This unusual formulation of evaluation contexts is intended to enforce determinism of the reduction relation. The idea is that evaluation never takes place inside or under a **let rec**, except the topmost one. Other bindings inside the

$$\frac{e \rightsquigarrow_c e'}{\mathbb{E}[e] \dashrightarrow_c \mathbb{E}[e']} \quad (\textsc{Context}) \qquad \frac{\mathbb{E}[\mathbb{N}](x) = v}{\mathbb{E}[\mathbb{N}[x]] \dashrightarrow_c \mathbb{E}[\mathbb{N}[v]]} \quad (\textsc{Subst})$$

$$\frac{\mathbf{dom}(b_1) \perp \{x\} \cup \mathbf{dom}(b_v, b_2) \cup \mathbf{FV}(b_v, b_2) \cup \mathbf{FV}(f)}{\textbf{let rec } b_v, x = (\textbf{let rec } b_1 \textbf{ in } e), b_2 \textbf{ in } f \rightsquigarrow_c \textbf{let rec } b_v, b_1, x = e, b_2 \textbf{ in } f} \quad (\text{IM})$$

$$\frac{\mathbf{dom}(b) \perp (\mathbf{dom}(b_v) \cup \mathbf{FV}(b_v))}{\textbf{let rec } b_v \textbf{ in let rec } b \textbf{ in } e \dashrightarrow_c \textbf{let rec } b_v, b \textbf{ in } e} \quad (\text{EM})$$

Figure 8: Reduction relation

expression first have to be lifted to the top by rule (Lift), and then merged with the topmost **let rec** if any, by rules (EM) and (IM). In the case where the topmost binding is of the shape $b_v, x = (\textbf{let rec } b_1 \textbf{ in } e), b_2$, rule (IM) allows to merge b_1 with the current binding. When an inner binding has been lifted to the top, if there is already a topmost binding, then the two bindings are merged together by rule (EM). As a result, when the evaluation encounters a binding, it is always possible to lift it up to the top and then merge it with the topmost binding if any.

Eventually, rule (Subst) describes the use of bound values when needed. The notion of a needed value is formalized by need contexts, which are defined by

$$\mathbb{N} ::= op[\Box] \mid \Box + v_1 \mid v_2 + \Box \qquad (v_2 \text{ is not a variable}).$$

In **MM** the value of a variable is copied only when needed for the application of an operator, or for composition. 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 in figure 7. There are two kinds of accesses.

- In the case of a context of the shape $\textbf{let rec } b_v \textbf{ in } \mathbb{F}$, if the called variable x is bound in the topmost binding b_v, then $b_v(x)$ is the requested value, provided the two capture conditions are respected. First, no variable free in $b_v(x)$ should be captured by $\mathbb{F}$. Second, x should not be captured by $\mathbb{F}$ either, because this would mean that another binding is concerned, inside $\mathbb{F}$.
- In the case of a context of the shape $\mathbb{E}[\textbf{let rec } b_v, y = \mathbb{F}, b \textbf{ in } e]$, if the called variable x is bound in the binding b_v, then $b_v(x)$ is the requested value, provided the two capture conditions are respected. First, no variable free in $b_v(x)$ should be captured by $\mathbb{F}$. Second, x should not be captured by $\mathbb{F}$ either, because this would mean that another binding is concerned, inside $\mathbb{F}$.

In figure 7, the capture conditions are formalized with the **Capt** function. $\mathbf{Capt}_{\Box}(\mathbb{E})$ is the set of bound variables above $\Box$ in $\mathbb{E}$. If $\Box$ is filled with another variable, then it is free in the obtained expression.

Instantiation The (Close) rule makes use of a reordering operation on outputs $\overline{o}$, which we define in this section. This operation takes four aspects of its argument into account: its internal dependencies, its fake dependencies, the shapes of its definitions, and its original ordering. Internal dependencies and fake dependencies are considered imperative requirements on the final ordering: if a definition d might call another definition d', then d' must be put before d in the final ordering. The shapes of the definitions are examined in order not to generate a binding with forward references to definitions of unpredictable shape. The original ordering is only used as a hint, in the case where no constraint forces one definition to be put before the other.

Remark 1 (Warning) *The criterion on bindings mentioned in section 2, forbidding forward dependency paths starting with a strict edge, will look reversed here. Indeed, when a definition d_1 calls another definition d_2, it is also possible to see it as a constraint on their ordering, such as "the definition d_2 must be put before the definition d_1". As we will use this relation on definitions as an ordering for generating a binding, the second way is more intuitive. A consequence is that the criterium now forbids backward dependency paths ending with a strict edge.*

$$\frac{(L[y^*] \triangleright x = e) \in o \qquad (L'[z^*] \triangleright x' = e') \in o \qquad \chi = \mathbf{Degree}(x', e)}{x' \xrightarrow{\chi}_o x}$$

$$\frac{(L[y_1 \dots y_n] \triangleright x = e) \in o \qquad (L'[z^*] \triangleright x' = e') \in o}{x' \xrightarrow{\text{☹}}_o x}$$

Figure 9: Dependencies in an output

$$\frac{N_1 \xrightarrow{\chi_1}{}^{+} N_2 \qquad N_2 \xrightarrow{\chi_2} N_3}{N_1 \xrightarrow{\chi_2}{}^{+} N_3} \qquad\qquad \frac{N_1 \xrightarrow{\chi} N_2}{N_1 \xrightarrow{\chi}{}^{+} N_2}$$

Figure 10: Transitive closure of $\rightarrow$

More formally, the dependency graph of an output is defined in figure 9. For each pair of definitions $L[y^*] \triangleright x = e$ and $L'[z^*] \triangleright x' = e'$ in o, there may be two kinds of edges.

- If x' is free in e, then an edge is drawn from x' to x. This edge is labeled with a degree $\chi \in \{\text{☺}, \text{☹}\}$. χ is determined by $\mathbf{Degree}(x', e)$, where the **Degree** function is defined for $x \in \mathbf{FV}(e)$ by

$$\begin{aligned} \mathbf{Degree}(x, \langle \iota \mid o \rangle) &= \text{☺} \\ \mathbf{Degree}(x, \{s_v\}) &= \text{☺} \\ \mathbf{Degree}(x, e) &= \text{☹ otherwise.} \end{aligned}$$

  The **Degree** function is simple, and could be extended as in [3, 14].

- If x' is mentioned in y^*, then an edge from x' to x is drawn, with degree ☹. Fake dependencies act as real strict dependencies.

The transitive closure of this relation is defined in figure 10, by defining the degree of a path as the degree of its last edge. The relation $\xrightarrow{\text{☹}}{}^{+}_o$ gives a conservative approximation of which definition needs the value of which other one in $\mathbf{Bind}(\overline{o})$. Reordering o according to $\rightarrow_o$ it is not enough though, because the generated binding might be syntactically incorrect. Indeed, it is forbidden to make forward references to definitions of unpredictable shape inside a binding. Strict forward references to definitions of unpredictable shape already correspond to edges labeled ☹ in $\rightarrow_o$, and are therefore taken into account when reordering according to $\xrightarrow{\text{☹}}{}^{+}_o$. Weak forward references to definitions of unpredictable shape correspond to edges labeled ☺ in $\rightarrow_o$, and are therefore not taken into account when reordering according to $\xrightarrow{\text{☹}}{}^{+}_o$. Let $\succ_o = \{(x_1, x_2) \mid x_1 \xrightarrow{\text{☺}} x_2, o(x_1) \notin \mathbf{Predictable}\}$. This relation exactly puts weak references to definitions of unpredictable shape in the right order.

We define the binary relation $\geqslant_o$ by the lexical ordering $\geqslant_o = \left((\xrightarrow{\text{☹}}{}^{+}_o \cup \succ_o)^+, >_o\right)$, where $>_o$ is the initial ordering in o. If $\geqslant_o$ contains no cycle, o is said correct. This is written $\vdash o$. In this case, $\overline{o}$ denotes o reordered by $\geqslant_o$.

4 Typing

4.1 Type system

In this section, we present a type system for **MM**.

Types are defined in figure 11. There are only two kinds of types, record types $\{O\}$ and mixin types $\langle I; O; D \rangle$, where I and O range over finite maps from names to types and D is a finite graph over names, labeled by degrees. Such a graph is called an abstract dependency graph. (Remember that dependency graphs over the whole set of nodes are called concrete.) An environment Γ is a finite map from variables to types. We write $\Gamma\langle\Gamma'\rangle$ for the map where the bindings of Γ' have overridden the ones from Γ.

$T \in \mathbf{Types}$	$::=$	$\{O\} \mid \langle I; O; D\rangle$
I, O	$\in$	$\mathbf{Names} \xrightarrow{\text{Fin}} \mathbf{Types}$
D	$\subset_{\text{Fin}}$	$\{X \xrightarrow{\chi} Y \mid X, Y \in \mathbf{Names}, \chi \in \mathbf{Degrees}\}$
Γ	$\in$	$\mathbf{Vars} \xrightarrow{\text{Fin}} \mathbf{Types}$

Figure 11: Types

Remark 2 *Graphs are considered equal modulo removal of isolated nodes, and modulo the following rewriting rule:*

$$N_1 \underset{\chi_2}{\overset{\chi_1}{\rightleftarrows}} N_2 \qquad - - \succ \qquad N_1 \xrightarrow{\chi_1 \wedge \chi_2} N_2 \tag{1}$$

where $\wedge$ gives the most dangerous of two degrees:

$$\chi_1 \wedge \chi_2 = \text{☺} \text{ if } \chi_1 = \chi_2 = \text{☺}$$
$$\chi_1 \wedge \chi_2 = \text{☹} \text{ otherwise}$$

In figure 12, the type system is defined by means of a set of inference rules.

The first rule (T-Struct) concerns the typing of basic structures $\langle \iota \mid o\rangle$. Given an input I (which is arbitrary here, we do not consider type inference or type-checking issues) corresponding to ι, and a type environment Γ_o correponding to o, it checks that the definitions in o indeed have the types mentioned in Γ_o.

The condition $\vdash \rightarrow_{\langle \iota | o\rangle}$ requires some explanation. We saw in section 3.2 that dependencies in an output are represented by its dependency graph $\rightarrow_o$. For structures (which are incomplete outputs), the corresponding notion is the concrete dependency graph. A concrete dependency graph is a graph over nodes. A node N is an element of $\mathbf{Nodes}(=)\,\mathbf{Vars} \cup \mathbf{Names}$. The dependency graph of a structure is defined in figure 14. It records dependencies in the structure (as was done for outputs), but takes external names into account, when possible. Named definitions are represented by a name, and local definitions are represented by their variables. In order for types not to mention local components, we introduce a *lift* operation $\lfloor \rightarrow_{\langle \iota | o\rangle} \rfloor$, which, as described in figure 13, first ensures to keep track of local components by shifting their dependencies to the next exported components, and then erases them. The result is an abstract dependency graph.

Finally, the rule checks that the imported types are well-formed, which would otherwise not be forced, with the following notion of well-formedness.

Definition 1 (Correct graphs) *A graph $\rightarrow$ is correct iff $\xrightarrow{\text{☹}}{}^{+}$ is an ordering on its nodes (written $\vdash \rightarrow$).*

Definition 2 (Well-formed types) *Figure 15 defines the sets of well-formed types an inputs (or outputs), as the least relation respecting the rules. A mixin type $\langle I; O; D\rangle$ must import and define disjoint sets of names, the targets of D must be defined, and D must be correct.*

The second rule (T-Sum) types the sum of two expressions. It verifies that names are bound to the same types in both expressions (relation $\square$ overloaded to types), that the union of the two dependency graphs is still correct, and that two names are not defined twice (i.e. are not in the two outputs). The result type shares the inputs, where defined names have been removed, and takes the union of the outputs and of the dependency graphs.

The third rule (T-Freeze) introduces a new operation $D!X \triangleright x$ on abstract graphs, which is again defined in figure 13. To freeze a name X, it first replaces X with a fresh local variable x, making the graph temporarily non-abstract. Then, it adds a strict link from x to X. This follows closely the semantics of freezing from figure 5, first making all other components call the local component x instead of X, and then re-exporting X as x exactly. The link is forced to be a strict one by hypothesis 2.

The (T-Close) rule transforms a mixin type with no input into a record type. It looks very simple, but to prove it correct, we must show that well-ordered outputs yield well-ordered bindings by contraction rule (Close).

The mixin projection rule (T-Project), exactly as the corresponding contraction rule, keeps in the output types only the selected ones, reporting the other ones in the input types. The abstract graph is modified

Expressions:

$$\frac{\mathbf{dom}(\iota) = \mathbf{dom}(I) \qquad \vdash I \qquad \vdash \Gamma_o \qquad \vdash \rightarrow_{\langle \iota | o \rangle} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \iota \mid o \rangle : \langle I; \Gamma_o \circ \mathbf{Input}(o); \lfloor \rightarrow_{\langle \iota | o \rangle} \rfloor \rangle} \quad \text{(T-Struct)}$$

$$\frac{I_1 \uplus O_1 \sqcap I_2 \uplus O_2 \qquad \vdash D_1 \cup D_2 \qquad \Gamma \vdash e_1 : \langle I_1; O_1; D_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2; O_2; D_2 \rangle}{\Gamma \vdash e_1 + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \uplus O_2; D_1 \cup D_2 \rangle} \quad \text{(T-Sum)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle \qquad X \in \mathbf{dom}(O)}{\Gamma \vdash e \,!\, X : \langle I; O; \lfloor D \,!\, X \rfloor \rangle} \quad \text{(T-Freeze)} \qquad \frac{\Gamma \vdash e : \langle \emptyset; O; D \rangle}{\Gamma \vdash \nabla e : \{O\}} \quad \text{(T-Close)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle}{\Gamma \vdash e_{|X_1 \ldots X_n} : \langle I \uplus O_{\setminus \{X_1 \ldots X_n\}}; O_{| \{X_1 \ldots X_n\}}; D_{| \{X_1 \ldots X_n\}} \rangle} \quad \text{(T-Project)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle}{\Gamma \vdash e_{|-X_1 \ldots X_n} : \langle I \uplus O_{| \{X_1 \ldots X_n\}}; O_{\setminus \{X_1 \ldots X_n\}}; D_{|-\{X_1 \ldots X_n\}} \rangle} \quad \text{(T-Delete)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle \qquad \{X_1 \ldots X_n\} \subset \mathbf{dom}(O)}{\Gamma \vdash e_{:-X_1 \ldots X_n} : \langle I; O_{\setminus \{X_1 \ldots X_n\}}; \lfloor D_{:-\{X_1 \ldots X_n\}} \rfloor \rangle} \quad \text{(T-Hide)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle \qquad \{X_1 \ldots X_n\} \subset \mathbf{dom}(O)}{\Gamma \vdash e_{:X_1 \ldots X_n} : \langle I; O_{| \{X_1 \ldots X_n\}}; \lfloor D_{:\{X_1 \ldots X_n\}} \rfloor \rangle} \quad \text{(T-Show)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle \qquad (\mathbf{cod}(r) \setminus \mathbf{dom}(r)) \perp (\mathbf{dom}(I) \cup \mathbf{dom}(O))}{\Gamma \vdash e[r] : \langle I\{r\}; O \circ \{r\}; D\{r\} \rangle} \quad \text{(T-Rename)}$$

$$\frac{\Gamma \vdash e : \langle I; O; D \rangle \qquad Y \notin \mathbf{dom}(I) \cup \mathbf{dom}(O)}{\Gamma \vdash e_{X \succ Y} : \langle I \uplus \{X : O(X)\}; O\{X \mapsto Y\}; D_{X \succ Y} \rangle} \quad \text{(T-Split)}$$

$$\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash e_i : T_i}{\Gamma \vdash \{X_1 = e_1 \ldots X_n = e_n\} : \{X_1 : T_1 \ldots X_n : T_n\}} \quad \text{(T-Record)} \qquad \frac{\Gamma \vdash e : \{O\}}{\Gamma \vdash e.X : O(X)} \quad \text{(T-Rselect)}$$

$$\frac{\vdash b \qquad \vdash \Gamma_b \qquad \Gamma\langle \Gamma_b \rangle \vdash b : \Gamma_b \qquad \Gamma\langle \Gamma_b \rangle \vdash e : T}{\Gamma \vdash \textbf{let rec } b \textbf{ in } e : T} \quad \text{(T-LetRec)} \qquad \frac{x \in \mathbf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-Variable)}$$

Sequences:

$$\Gamma \vdash \epsilon : \emptyset \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash o : \Gamma_o}{\Gamma \vdash (L[x^*] \triangleright x = e, o) : \{x : T\} \uplus \Gamma_o} \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x = e, b) : \{x : T\} \uplus \Gamma_b}$$

Figure 12: Type system

Lift

Transitive closure through local components

$$\frac{N_1 \xrightarrow{\chi_1} x \qquad x \xrightarrow{\chi_2}^{\Box} N_2}{N_1 \xrightarrow{\chi_1 \wedge \chi_2}^{\Box} N_2} \qquad \frac{N_1 \xrightarrow{\chi} N_2}{N_1 \xrightarrow{\chi}^{\Box} N_2}$$

Lift

	$\lfloor \rightarrow \rfloor$	$=$	$\rightarrow^{\Box}_{\mid \mathbf{Names} \times \mathbf{Names}}$
Sum	$D_1 + D_2$	$=$	$D_1 \cup D_2$
Freeze	$D \,!\, X$	$=$	$D\{X \leftarrow x\} \cup \{x \xrightarrow{\frowny} X\}$ (x not mentioned in D)
Project	$D_{\mid \mathcal{N}}$	$=$	$D_{\mid \mathbf{Names} \times \mathcal{N} \times \mathbf{Degrees}}$
Delete	$D_{\mid -\mathcal{N}}$	$=$	$D_{\setminus \mathbf{Names} \times \mathcal{N} \times \mathbf{Degrees}}$
Hide	$D_{:-X_1 \ldots X_n}$	$=$	$D\{X_1 \mapsto x_1 \ldots X_n \mapsto x_n\}$ ($x_1 \ldots x_n$ fresh)
Show	$D_{:X_1 \ldots X_n}$	$=$	$D_{:- \mathbf{Targets}(D) \setminus \{X_1 \ldots X_m\}}$
Rename	$D\{r\}$	$=$	$\{(N_1\{r\}, N_2\{r\}, \chi) \mid (N_1, N_2, \chi) \in D\}$
Split	$D_{X \succ Y}$	$=$	$(D \setminus D_{\mid \{. \rightarrow X\}}) \cup \{(Z, Y, \chi) \mid (Z, X, \chi) \in D\}$

Figure 13: Graph operations

$$\frac{\chi = \mathbf{Degree}(x', e) \qquad (L', x') \in \mathbf{dom}(\langle \iota \mid o \rangle) \qquad (L[z^*] \triangleright x = e) \in o}{\mathbf{Node}(L', x') \xrightarrow{\chi}_{\langle \iota \mid o \rangle} \mathbf{Node}(L, x)}$$

$$\frac{(L_i, x_i) \in \mathbf{dom}(\langle \iota \mid o \rangle) \qquad (L[z_1 \ldots z_n] \triangleright x = e) \in o}{\mathbf{Node}(L_i, x_i) \xrightarrow{\frowny}_{\langle \iota \mid o \rangle} \mathbf{Node}(L, x)}$$

Figure 14: Dependencies in a structure

$$\frac{\vdash I \qquad \vdash O \qquad \mathbf{dom}(I) \perp \mathbf{dom}(O) \qquad \mathbf{Targets}(D) \subset \mathbf{dom}(O) \qquad \vdash D}{\vdash \langle I; O; D \rangle} \qquad \frac{\vdash O}{\vdash \{O\}}$$

$$\frac{\forall X \in \mathbf{dom}(I) \vdash I(X)}{\vdash I}$$

Figure 15: Well-formed types

accordingly by the operation $D_{|\{X_1 \dots X_n\}}$, which removes the edges leading to unselected components. The (T-DELETE) rule is its dual again.

The (T-HIDE) removes the given names from the output. Additionally, it acts on the abstract graph D as described in figure 13. It first replaces the given names by fresh variables, and then lifts the result, in order to obtain an abstract graph. Rule (T-SHOW) is its dual, as expected.

Rule (T-RENAME), given a mixin e of type $\langle I; O; D\rangle$, deduces that e renamed by r has the same type, with input I and output O redirected to use the new names ($\mathbf{cod}(r)$). As the contraction rule (RENAME), it makes use of the $r_{\mathcal{N}}$ function, composed with I and O. The abstract graph is renamed as well.

Given an expression e of type $\langle I; O; D\rangle$, according to rule (T-SPLIT), the type of $e_{X \succ Y}$ is as follows. X is added to the input, with the type it had in O. X is renamed to Y in the output. The graph D is modified according to figure 13. $D_{|\{. \to X\}}$ is the set of edges leading to X in D. Basically, these edges are redirected to Y.

The (T-RSELECT) and (T-RECORD) rules for typing record construction and selection are standard.

The (T-LETREC) for typing bindings **let rec** b **in** e is almost standard, except for its side-condition: the binding must be well-ordered with respect to its dependencies. The dependency graph of a binding b is defined via the dependency graph of the equivalent output $\mathbf{Output}(b) = \mathbf{Output}(x_1 = e_1 \dots x_n = e_n) = (\_[] \triangleright x_1 = e_1 \dots \_[] \triangleright x_n = e_n)$. We define $\geqslant_b$ by $\geqslant_{\mathbf{Output}(b)}$. A binding b is said correct with respect to an ordering $>$ (written $>\vdash b$) if $>_b$ (the definition order in b) respects $>$ (in other words $>\subset>_b$). We abbreviate $\geqslant_b \vdash b$ with $\vdash b$.

Eventually, the typing of outputs and bindings is straightforward, since it consists in successively typing their definitions.

4.2 Graph soundness

In section 3, we presented **MM** with concrete, simple instances of **IsDefinedSize**() and **Degree**. We now axiomatize the minimum conditions that they must satisfy.

Hypothesis 1 (Shape)

- $x \notin$ **Predictable**.
- $\langle \iota \mid o\rangle \in$ **Predictable** *and* $\{s_v\} \in$ **Predictable**.
- *Let* σ *be a variable renaming.* $e\{\sigma\} \in$ **Predictable** *iff* $e \in$ **Predictable**.
- *If* $\mathbb{E}[x] \in$ **Predictable**, *then* $\mathbb{E}[v] \in$ **Predictable**, *for all* v.
- *If* $e \longrightarrow e'$ *and* $e \in$ **Predictable**, *then* $e' \in$ **Predictable**.
- *If* $e \in$ **Predictable** *and* $e' \in$ **Predictable**, *then for any context* $\mathbb{E}$, $\mathbb{E}[e] \in$ **Predictable** *iff* $\mathbb{E}[e'] \in$ **Predictable**.

We require the degree function to meet the following condition.

Hypothesis 2 (Degree function)

- *If* $\mathbf{Degree}(x, e) = \odot$, *then* $e \in$ **Predictable**.
- *If* $e \longrightarrow e'$ *and* $\mathbf{Degree}(x, e) \neq \circledcirc$, *then* $\mathbf{Degree}(x, e') \neq \circledcirc$.
- *If* $x \in \mathbf{FV}(e) \setminus \mathbf{Capt}_{\square}(\mathbb{E}[\mathbb{N}])$, *then* $\mathbf{Degree}(x, \mathbb{E}[\mathbb{N}[e]]) = \circledcirc$.
- *If* $y \notin \mathbf{FV}(v) \setminus \mathbf{Capt}_{\square}(\mathbb{F})$, *then* $\mathbf{Degree}(y, \mathbb{F}[v]) = \mathbf{Degree}(y, \mathbb{F})$.
- *If for all* $x \in \mathbf{FV}(e)$, $\mathbf{Degree}(x, e') \leq \mathbf{Degree}(x, e)$, *then for any context* $\mathbb{E}$, *for any* $x \in \mathbf{FV}(\mathbb{E}[e])$, $\mathbf{Degree}(x, \mathbb{E}[e']) \leq \mathbf{Degree}(x, \mathbb{E}[e])$.
- $\forall x \notin \mathbf{dom}(b), X \neq Y, \forall \chi \in \{\chi \mid X \xrightarrow{\chi}_{\langle X \triangleright x \mid o\rangle} N, o = (\mathbf{Output}(b), Y \triangleright \_ = e)\}$,

$$\mathbf{Degree}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) \leq \chi.$$

4.2.1 Modeling the reduction with graphs

Definition 3 (Mixin redex) *Mixin redexes $e_\uparrow$ are defined by*

$$e_\uparrow ::= \langle \iota_1 \mid o_1 \rangle + \langle \iota_2 \mid o_2 \rangle \mid op[\langle \iota \mid o \rangle].$$

The graph operations on abstract graphs defined in figure 13 are trivially generalized to concrete graphs. These operations are used to guess the concrete graph of a mixin redex.

Definition 4 (Graph of a mixin redex)

$$\begin{array}{lcl} \rightarrow_{\langle \iota_1 \mid o_1 \rangle + \langle \iota_2 \mid o_2 \rangle} & = & \rightarrow_{\langle \iota_1 \mid o_1 \rangle} + \rightarrow_{\langle \iota_2 \mid o_2 \rangle} \\ \rightarrow_{op[\langle \iota \mid o \rangle]} & = & op(\rightarrow_{\langle \iota \mid o \rangle}) \end{array}$$

Proposition 1 (Graphs operations model contraction) *If $e_\uparrow \rightsquigarrow_c e$, then $\rightarrow_{e_\uparrow} = \rightarrow_e$.*

Proof By case analysis on the reduction.

Sum. We have $e_\uparrow = \langle \iota_1 \mid o_1 \rangle + \langle \iota_2 \mid o_2 \rangle$ and $e = \langle \iota \mid o_1, o_2 \rangle$, with $\iota = (\iota_1 \cup \iota_2) \setminus \mathbf{Input}(o_1, o_2)$. Trivially, $\rightarrow_{e_\uparrow} = \rightarrow_{\langle \iota_1 \mid o_1 \rangle} \cup \rightarrow_{\langle \iota_2 \mid o_2 \rangle} = \rightarrow_{\langle \iota \mid o_1 \rangle} \cup \rightarrow_{\langle \iota \mid o_2 \rangle}$ by $\langle \iota_1 \mid o_1 \rangle \square \langle \iota_2 \mid o_2 \rangle$. Then, $\rightarrow_{\langle \iota \mid o_1 \rangle} \cup \rightarrow_{\langle \iota \mid o_2 \rangle} = \rightarrow_{\langle \iota \mid o_1, o_2 \rangle}$.

Freeze. Let $e_\uparrow = \langle \iota \mid o_1, X[y^*] \triangleright x = f, o_2 \rangle \,!\, X$, and $e = \langle \iota \mid o_1, \_[y^*] \triangleright x = f, o_2, X \triangleright \_ = x \rangle$. First consider the structure $e' = \langle \iota \mid o_1, \_[y^*] \triangleright x = f, o_2 \rangle$. Its graph is exactly the same as the one of e except that instead of the node X, we find the node $\mathbf{Node}(\_ \triangleright x)$, which is x. Then, append the component $X \triangleright y = x$ with a fresh y. This adds a strict dependency from X to x, so the result is exactly $\rightarrow_{e_\uparrow}$.

Other cases similar.

□

4.2.2 Subject contraction for graphs

The goal of this section is to ensure that abstract graphs detect all errors in the underlying concrete graphs. We write γ for paths in graphs. The minimum degree of a path $\gamma = X \xrightarrow{\chi_1} N_1 \dots N_{n-1} \xrightarrow{\chi_n} Y$ is $\chi = \bigwedge_{1 \le i \le n} \chi_i$.

Proposition 2 (Lift preserve paths between names) *Let γ be a path for the $\rightarrow$ relation, starting with name X, ending with name Y, and having minimum degree χ. Let $D = \lfloor \rightarrow \rfloor$. There exists a path from X to Y in D, with the same minimum degree.*

Proof Let $\gamma = N_0 \xrightarrow{\chi_1} N_1 \dots N_{n-1} \xrightarrow{\chi_n} N_n$. We proceed by induction on the number of names in the path.

Base. Two names, $\gamma = X \xrightarrow{\chi_1} x_1 \dots x_{n-1} \xrightarrow{\chi_n} Y$. An easy induction on n proves that $X \xrightarrow{\chi}{}^{\square} Y$, and therefore $(X, Y, \chi) \in D$.

Induction. By induction hypothesis.

□

Corollary 1 *If $\xrightarrow{\frowny}{}^+$ has a cycle with at least one name, then $\lfloor \rightarrow \rfloor^{\frowny +}$ also has one.*

On the other hand, lifting commutes with the other operations on graphs.

Proposition 3 (Lift commutes with operators) *Let $\rightarrow_1$, $\rightarrow_2$, and $\rightarrow$ be concrete dependency graphs (i.e. graphs over* **Nodes***).*

- *If the variables from $\rightarrow_1$ and the ones from $\rightarrow_2$ are disjoint, then $\lfloor \rightarrow_1 \cup \rightarrow_2 \rfloor = \lfloor \rightarrow_1 \rfloor \cup \lfloor \rightarrow_2 \rfloor$.*
- *$\lfloor op[\rightarrow] \rfloor = op[\lfloor \rightarrow \rfloor]$, for $op \in \{!X, \mid\!-_{\mathcal{N}}, \mid_{\mathcal{N}}, [r], :\!-_{\mathcal{N}}, :_{\mathcal{N}}, X \succ Y\}$ (with $\mathbf{cod}(r) \perp \mathbf{Nodes}(\rightarrow)$).*

Proof

Sum. It is obvious that $\lfloor \to_1 \rfloor \cup \lfloor \to_2 \rfloor \subset \lfloor \to_1 \cup \to_2 \rfloor$, since $\to_1 \subset \to_1 \cup \to_2$ and lift is monotone.

Now, an edge between names X and Y in $\lfloor \to_1 \cup \to_2 \rfloor$, implies the existence of a path between X and Y through variables only, in $\to_1 \cup \to_2$, but as variables cannot interact, this path is entirely in either one of the two subgraphs.

Freeze. Let x be a fresh variable. By definition, we have to prove that $\lfloor \to !X \triangleright x \rfloor = \lfloor \lfloor \to \rfloor !X \triangleright x \rfloor$. Let

$$\begin{array}{lcllcl} \to_1 & = & \to !X \triangleright x & \text{and} & \to_2 & = & \lfloor \to \rfloor \\ \to_1' & = & \lfloor \to_1 \rfloor & & \to_2' & = & \to_2 !X \triangleright x \\ & & & & \to_2'' & = & \lfloor \to_2' \rfloor \end{array}$$

First, notice that both in $\to_1'$ and $\to_2''$, no edge starts from X, and edges arriving to X come from paths to X through x with degree ☹ in $\to_1$ and $\to_2'$, respectively, so they have degree ☹.

- $\to_2'' \subset \to_1'$.
  - Let $Y \xrightarrow{\chi}{}_2'' X$, with $X \neq Y$. Necessarily, $\chi = ☹$.
    This implies that there exists a path of $\to_2'$ of the shape
    $$Y \xrightarrow{\chi_0}{}_2' x \xrightarrow{\chi_1}{}_2' x \ldots \xrightarrow{\chi_n}{}_2' x \xrightarrow{☹}{}_2' X,$$
    because x is the only variable in $\to_2'$.
    n could be zero, in which case we would have $Y \xrightarrow{☹}{}_2' X$.
    But this means that we have $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \ldots \xrightarrow{\chi_n}{}_2 X$.
    So by definition of $\lfloor\rfloor$, we have $Y \xrightarrow{\chi_0}{}^{\square} X \xrightarrow{\chi_1}{}^{\square} X \ldots \xrightarrow{\chi_n}{}^{\square} X$.
    So, we have $Y \xrightarrow{\chi_0}{}_1^{\square} x \xrightarrow{\chi_1}{}_1^{\square} x \ldots \xrightarrow{\chi_n}{}_1^{\square} x \xrightarrow{☹}{}_1 X$,
    and therefore $Y \xrightarrow{☹}{}_1' X$.
  - Let $Y \xrightarrow{\chi}{}_2'' Z$, with Y and Z different from X. Then
    $$Y \xrightarrow{\chi_0}{}_2' x \xrightarrow{\chi_1}{}_2' x \ldots x \xrightarrow{\chi_n}{}_2' Z,$$
    because x is the only variable in $\to_2'$.
    We have $\chi = \bigwedge_{0 \leq i \leq n} \chi_i$. n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}_2' Z$.
    We can deduce $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \ldots X \xrightarrow{\chi_n}{}_2 Z$,
    so $Y \xrightarrow{\chi_0}{}^{\square} X \xrightarrow{\chi_1}{}^{\square} X \ldots X \xrightarrow{\chi_n}{}^{\square} Z$,
    and therefore $Y \xrightarrow{\chi_0}{}_1^{\square} x \xrightarrow{\chi_1}{}_1^{\square} x \ldots x \xrightarrow{\chi_n}{}_1^{\square} Z$.
    So we have $Y \xrightarrow{\chi}{}_1' Z$.
- $\to_1' \subset \to_2''$.
  - Let $Y \xrightarrow{\chi}{}_1' X$, with $Y \neq X$. We have
    $$Y \xrightarrow{\chi_0}{}_1^{\square} x \xrightarrow{\chi_1}{}_1^{\square} x \ldots x \xrightarrow{\chi_n}{}_1^{\square} x \xrightarrow{☹}{}_1 X,$$
    where for all i, $\xrightarrow{\chi_i}{}_1^{\square}$ does not go through x.
    As above, we have $\chi = ☹$ and n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}_1^{\square} x \xrightarrow{☹}{}_1 X$.
    This implies that $Y \xrightarrow{\chi_0}{}^{\square} X \xrightarrow{\chi_1}{}^{\square} X \ldots X \xrightarrow{\chi_n}{}^{\square} X$.
    Therefore, $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \ldots X \xrightarrow{\chi_n}{}_2 X$.
    So, $Y \xrightarrow{\chi_0}{}_2' x \xrightarrow{\chi_1}{}_2' x \ldots x \xrightarrow{\chi_n}{}_2' x \xrightarrow{☹}{}_2' X$,
    and so $Y \xrightarrow{☹}{}_2'' X$.
  - Let $Y \xrightarrow{\chi}{}_1' Z$, with Y and Z different from X.
    We deduce $Y \xrightarrow{\chi_0}{}_1^{\square} x \xrightarrow{\chi_1}{}_1^{\square} x \ldots x \xrightarrow{\chi_n}{}_1^{\square} Z$,
    where for all i, $\xrightarrow{\chi_i}{}_1^{\square}$ does not go through x.

As above, we have $\chi = \bigwedge_{0 \leq i \leq n} \chi_i$ and n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}^{\square}_{1} Z$.
Then, $Y \xrightarrow{\chi_0}{}^{\square} X \xrightarrow{\chi_1}{}^{\square} X \ldots X \xrightarrow{\chi_n}{}^{\square} Z$,
and so $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \ldots X \xrightarrow{\chi_n}{}_2 Z$,
which leads to $Y \xrightarrow{\chi_0}{}'_2 x \xrightarrow{\chi_1}{}'_2 x \ldots x \xrightarrow{\chi_n}{}'_2 Z$,
and so $Y \xrightarrow{\chi}{}''_2 Z$.

Other cases. Easy.

□

Corollary 2 *If* $\vdash \rightarrow$, *then* $\vdash op(\rightarrow)$. *If* $\vdash \rightarrow_1$, $\vdash \rightarrow_2$, **Variables**$(\rightarrow_1) \perp$ **Variables**$(\rightarrow_2)$, *and* $\vdash \lfloor \rightarrow_1 \rfloor \cup \lfloor \rightarrow_2 \rfloor$, *then* $\vdash \rightarrow_1 \cup \rightarrow_2$.

Proof

Freeze. Assume $op = !X \triangleright x$ and $G = (A, >, \rightarrow)$. This operation first replaces X by x in $\rightarrow$, which does not introduce any cycle, and then adds one-way edges to X, which cannot create any cycle.

Sum. Let $\rightarrow = \rightarrow_1 \cup \rightarrow_2$. Assume there is a cycle in $\xrightarrow{\text{☹}}{}^{+}$. First notice that if there were no named node in it, as variables from both graphs do not interact, the cycle would come entirely from one of the two graphs, which are supposed correct, therefore contradicting the hypothesis. Otherwise, by lemma 3, $\lfloor \rightarrow \rfloor = \lfloor \rightarrow_1 \rfloor \cup \lfloor \rightarrow_2 \rfloor$. Moreover, there is at least one named node X in our cycle, so by lemma 2, our cycle is a path from X to X, so it appears in $\lfloor \rightarrow \rfloor$ with the same valuation, which contradicts its correctness.

Other cases. Easy, since they do not add any edge to the dependencies.

□

Proposition 4 *If* $\Gamma \vdash e_\uparrow : \langle I; O; D \rangle$, *then* $D = \lfloor \rightarrow_{e_\uparrow} \rfloor$.

As a consequence, if a mixin redex is well-typed, then the structure(s) in it have a correct graph, and by typing the redex also has a correct graph.

Corollary 3 *If* $\Gamma \vdash e_\uparrow : T$, *then* $\vdash \rightarrow_{e_\uparrow}$.

Lemma 1 *If* $\Gamma \vdash e_\uparrow : \langle I; O; D \rangle$ *and* $e_\uparrow \rightsquigarrow_c e$, *then* e *is a structure and* $\vdash \rightarrow_e$ *and* $D = \lfloor \rightarrow_e \rfloor$.

We have proven that structures obtained by reduction are correct, which means that their dependencies do not have strict cycles. It is now necessary to prove that this property is enough for a structure without inputs to be closed. In other words, it is necessary for our type system to be sound that an output with a correct dependency graph generate can be reordered.

Lemma 2 (Typing is enough for close) *If* $\vdash \rightarrow_o$, *then* $\vdash o$.

Proof Assume there is a cycle in $\succcurlyeq_o = (\succ_o \cup \xrightarrow{\text{☹}}{}^{+}_{o})^+$. This cycle cannot contain only $\succ_o$ edges, since for all nodes N_1, N_2, N_3 such that $N_1 \succ_o N_2 \succ_o N_3$, by definition $N_1 \xrightarrow{\text{☺}}{}_o N_2 \xrightarrow{\text{☺}}{}_o N_3$, with $o(N_1) \notin$ **Predictable** and $o(N_2) \notin$ **Predictable**, and by definition of $\succ_o$ and hypothesis 2, we have $o(N_2) \in$ **Predictable**, which is a contradiction.

So there is at least one $\xrightarrow{\text{☹}}{}^{+}_{o}$ edge in our cycle. But $\succ_o$ is included in $\xrightarrow{\text{☺}}{}_o$, so this is a cycle for $\xrightarrow{\text{☹}}{}^{+}_{o}$ too.
□

4.2.3 Manipulation of recursive bindings

Definition 5 (Graph comparison) *We define* $\rightarrow_1 < \rightarrow_2$ *by*

- *for all* $N_1 \xrightarrow{\text{☺}}_2 N_2$, *there exists* $N_1 \xrightarrow{\text{☺}}_1^+ N_2$
- *for all* $N_1 \xrightarrow{\text{☺}}_2 N_2$, *there exists* $N_1 \xrightarrow{\chi}_1 N_2$.

In particular, if $\rightarrow_2 \subset \rightarrow_1$, then $\rightarrow_1 < \rightarrow_2$; and if for all edge in $\rightarrow_2$ there exists an edge with the same ends and an inferior degree in $\rightarrow_1$, then $\rightarrow_1 < \rightarrow_2$. Notice that this relation is transitive.

Definition 6 (Binding comparison) *A binding* b_1 *is more restrictive than a binding* b_2 *(written* $b_1 < b_2$*) iff they have the same domains (*$\mathbf{dom}(b_1) = \mathbf{dom}(b_2)$*), they define variables in the same order (*$>_{b_1} = >_{b_2}$*), the dependencies and shapes of* b_1 *are more restrictive than those of* b_2 *(*$\rightarrow_{b_1} < \rightarrow_{b_2}$*, and for all* $x \in \mathbf{dom}(b_2)$*, if* $b_2(x) \notin \mathbf{Predictable}$*, then* $b_1(x) \notin \mathbf{Predictable}$*).*

The desired property is that if a binding is well-ordered for the ordering induced by a more restrictive binding, then it is well-ordered.

Lemma 3 (Relax) *If* $b' < b$ *and* $\geqslant_{b'} \vdash b$, *then* $\vdash b$.

Proof We proceed by contrapositive. First notice that $\geqslant_{b'} \vdash b$ implies $\geqslant_{b'} \vdash b'$, since they define variables in the same order. If $\geqslant_b \vdash b$ does not hold, it implies that there is a right-to-left edge in $(\succ_{\mathbf{Output}(b)} \cup \xrightarrow{\text{☹}}_b^+)^+$. So, there exists $x = e$ and $y = f$ defined in b in this order, such that either $y \succ_{\mathbf{Output}(b)} x$ or $y \xrightarrow{\text{☹}}_b^+ x$.

- If $y \succ_{\mathbf{Output}(b)} x$, then $y \xrightarrow{\text{☺}}_b x$ and $b(y) \notin \mathbf{Predictable}$. By definition of $b' < b$, this implies that $b'(y) \notin \mathbf{Predictable}$ and $y \xrightarrow{\chi}_{b'} x$. Whatever χ is, it is a right-to-left edge in $\geqslant_{b'}$, which contradicts $\vdash b'$.
- If $y \xrightarrow{\text{☹}}_b^+ x$, by definition of $b' < b$, this implies that $y \xrightarrow{\text{☹}}_{b'}^+ x$, so it is a right-to-left edge in $\geqslant_{b'}$, which contradicts $\vdash b'$.

□

Lemma 4 *If* $\vdash \rightarrow_{\langle \epsilon | o \rangle}$, *then* $\vdash \mathbf{Bind}(\overline{o})$.

Proof $\mathbf{Bind}(\overline{o})$ is in the same order as $\overline{o}$, and its graph does not take fake dependencies into account. Lemma 3 allows to conclude. □

Our computational reduction relation manipulates **let rec** constructs as blocks of data, not worrying too much about dependencies issues. The soundness proof requires some properties to be verified, especially concerning the (IM) rule, which merges two nested bindings. We want to be sure that merging two well-orderd internally nested bindings – i.e. the second binding appears in one of the definitions of the first one – yield a well-ordered new binding (corollary 4).

Definition 7 (Paths) *For a path* $\gamma = (N_0 \xrightarrow{\chi_1} \dots \xrightarrow{\chi_n} N_n)$, *we define the degree of* γ *as* χ_n, *and we write* γ^χ *for a path of degree* χ, *and* $\gamma \subset \rightarrow$ *if* γ *is a path of* $\rightarrow$.

Eventually, we write edges as triples (source, target, degree), and paths as lists of paths such that the target of one is the source of the next one, separated by commas, as in $\gamma_1^{\chi_1}, (x, y, \chi), \gamma_2^{\chi_2}$.

Proposition 5 (Let rec internal dependencies)
For all y, *for all* $x \in \mathbf{FV}(e) \setminus \mathbf{dom}(b)$, $\mathbf{Degree}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) \leq \mathbf{Degree}(x, e)$.
For all y, *for all* $x \in \mathbf{FV}(b(y)) \setminus \mathbf{dom}(b)$, $\mathbf{Degree}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) \leq \mathbf{Degree}(x, b(y))$.

Proof
Let $X \neq Y$, $b = (x_1 = e_1 \dots x_n = e_n)$, and $o = (\mathbf{Output}(b), Y \triangleright \_ = e)$.

- For the first point, as $x \in \mathbf{FV}(e)$, there is an edge $X \xrightarrow{\chi}_{\langle X \triangleright x | o \rangle} Y$, where $\chi = \mathbf{Degree}(x, e)$. By hypothesis 2, $\mathbf{Degree}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) \leq \chi$.

- The second point is similar. Suppose $y = x_{i_0}$ and $f = b(y)$. There is an edge $X \xrightarrow{\chi}_{\langle X \triangleright x \mid o\rangle} x_{i_0}$, where $\chi = \mathbf{Degree}(x, f)$. By hypothesis 2, $\mathbf{Degree}(x, \mathbf{let\ rec}\ b\ \mathbf{in}\ e) \leq \chi$.

□

Proposition 6 (Merging nested bindings)
Let $b = (b_1, x = \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e, b_3)$, $b' = (b_1, b_2, x = e, b_3)$, with $\vdash b$ and $\mathbf{dom}(b_2) \perp \mathbf{dom}(b) \cup \mathbf{FV}(b_1, b_3)$. Let γ a path of $\rightarrow_{b'}$, from x_1 to x_2, of degree χ.

1. *If $x_1, x_2 \in \mathbf{dom}(b)$, then $x_1 \xrightarrow{\chi'}{}^{+}_{b} x_2$, with $\chi' \leq \chi$.*
2. *If $x_1 \in \mathbf{dom}(b), x_2 \in \mathbf{dom}(b_2)$, then $x_1 \xrightarrow{\chi'}{}^{+}_{b} x$, with $\chi' \leq \chi$.*
3. *If $x_1 \in \mathbf{dom}(b_2), x_2 \in \mathbf{dom}(b)$, then if $\chi = \circledcirc$, then $x_2 \in (\{x\} \cup \mathbf{dom}(b_3))$.*
4. *If $x_1, x_2 \in \mathbf{dom}(b_2)$, then either $\gamma \subset \rightarrow_{b_2}$ or $x \xrightarrow{\chi'}{}^{+}_{b} x$ for some $\chi' \leq \chi$.*

Proof By induction on the length of γ.

Base γ is an edge.

1. $x_1, x_2 \in \mathbf{dom}(b)$. If $x_2 \neq x$, $\chi = \mathbf{Degree}(x_1, b'(x_2)) = \mathbf{Degree}(x_1, b(x_2))$, so $x_1 \xrightarrow{\chi}_{b} x_2$. Otherwise, $\chi = \mathbf{Degree}(x_1, e)$.
   But as $x_1 \notin \mathbf{dom}(b_2)$, by lemma 5, $\mathbf{Degree}(x_1, \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e) \leq \mathbf{Degree}(x_1, e)$, so we have an edge $x_1 \xrightarrow{\chi'}_{b} x$, with $\chi' \leq \chi$.
2. $x_1 \in \mathbf{dom}(b), x_2 \in \mathbf{dom}(b_2)$. Let $b_2(x_2) = f$. We have $\chi = \mathbf{Degree}(x_1, f)$, so similarly by lemma 5, $\chi' = \mathbf{Degree}(x_1, \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e) \leq \mathbf{Degree}(x_1, f)$, so we have an edge $x_1 \xrightarrow{\chi'}_{b} x$, with $\chi' \leq \chi$.
3. $x_1 \in \mathbf{dom}(b_2), x_2 \in \mathbf{dom}(b)$. We have $x_1 \in \mathbf{FV}(b'(x_2))$ and $x_2 \in \mathbf{dom}(b)$, so $x_2 = x$, so $x_2 \in (\{x\} \cup \mathbf{dom}(b_3))$.
4. $x_1, x_2 \in \mathbf{dom}(b_2)$, we have of course $\gamma \subset \rightarrow_{b_2}$.

Induction step γ is of length $n > 1$.

1. $x_1, x_2 \in \mathbf{dom}(b)$.
   - If γ only has nodes in $\mathbf{dom}(b)$, let (x_3, x_2, χ) be its last edge. By induction hypothesis there is a path $\gamma_1^{\chi'}$ from x_3 to x_2 with $\chi' \leq \chi$ in $\rightarrow_b$, and a path $\gamma_2^{\chi''}$ from x_1 to x_3, so $\gamma_1^{\chi''}, \gamma_2^{\chi'} \subset \rightarrow_b$, with degree $\chi' \leq \chi$.
   - Otherwise, let x_5 be the last node of γ in $\mathbf{dom}(b_2)$. The next node is necessarily x. Let x_3 be the last node of γ in $\mathbf{dom}(b)$ before x_5. Let x_4 be the next node. (It is in $\mathbf{dom}(b_2)$.) We have
   $$\gamma = \gamma_1^{\chi_1}, (x_3, x_4, \chi_4), \gamma_2^{\chi_2}, (x_5, x, \chi_5), \gamma_3^{\chi_3},$$
   with $\gamma_2^{\chi_2} \subset \rightarrow_{b_2}$. Let now $o = (\mathbf{Output}(b_2), Y \triangleright \_ = e)$ and consider the structure $\langle X \triangleright x_3 \mid o\rangle$. Its concrete dependency graph is $\rightarrow_{\langle X \triangleright x_3 \mid o\rangle}$ and contains a path $(X, x_4, \chi_4), \gamma_2^{\chi_2}, (x_5, Y, \chi_5)$.
   So by hypothesis 2, we have $\chi'_5 = \mathbf{Degree}(x_3, \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e) \leq \chi_5$, so there is and edge $x_3 \xrightarrow{\chi'_5}_{b} x$.
   Then, applying the induction hypothesis to γ_1 and γ_3 if not empty, we obtain two paths ${\gamma'_1}^{\chi'_1}$ and ${\gamma'_3}^{\chi'_3}$ of $\rightarrow_b$, and so ${\gamma'_1}^{\chi'_1}, (x_3, x, \chi'_5), {\gamma'_3}^{\chi'_3}$ is a path of $\rightarrow_b$, with a degree $\chi'_5 \leq \chi_5$ if γ_3 is empty, and a degree $\chi'_3 \leq \chi_3$ otherwise.
2. $x_1 \in \mathbf{dom}(b), x_2 \in \mathbf{dom}(b_2)$. Let x_3 be the last node of γ in $\mathbf{dom}(b)$, and x_4 the next one. γ is of the shape $\gamma_1^{\chi_1}, (x_3, x_4, \chi_3), \gamma_2^{\chi_2}$, with the nodes of γ_2 in $\mathbf{dom}(b_2)$. γ_1 and γ_2 could be empty. As above, by lemma 5, we have $\chi'_3 = \mathbf{Degree}(x_3, \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e) \leq \mathbf{Degree}(x_3, b_2(x_4)) = \chi_3$, so we have an edge $x_3 \xrightarrow{\chi'_3}_{b} x$.

- If γ_2 is empty, $n > 1$, so γ_1 is non-empty, and applying induction hypothesis to γ_1, we obtain ${\gamma'_1}^{\chi'_1}$ with same ends, and therefore obtain a path in $\rightarrow_b$ with same ends as γ, and with degree $\chi'_3 \leq \chi_3 = \chi$.
- Otherwise, ${\gamma_2}^{\chi_2} \subset \rightarrow_{b_2}$. Let $X \neq Y$, $\iota = X \triangleright x_3$, and $o = \mathbf{Output}(b_2), Y \triangleright \_ = e$. We obtain a path $(x_3, x_4, \chi_3), {\gamma_2}^{\chi_2}$ in $\rightarrow_{\langle \iota | o \rangle}$ with same ends as γ, and with degree $\chi_2 = \chi$. So, if γ_1 is empty, we have in both cases a path from x_3 to x in b, with degree $\chi'_2 \leq \chi$. Otherwise, by induction hypothesis, we obtain ${\gamma'_1}^{\chi'_1}$ with $\chi'_1 \leq \chi_1$, and reason exactly as above.

3. $x_1 \in \mathbf{dom}(b_2), x_2 \in \mathbf{dom}(b)$. Assume $\chi = \frownie$. The first node of γ not in $\mathbf{dom}(b_2)$ is necessarily x. Let x_3 be the node just before it. γ has the shape $\gamma_1^{\chi_1}, (x_3, x, \chi_3), \gamma_2^{\chi_2}$. If γ_2 is empty, we have $x_2 = x$ which is clearly in $\{x\} \cup \mathbf{dom}(b_3)$. Otherwise, apply induction hypothesis to obtain a path ${\gamma'_2}^{\chi'_2}$ with the same ends as γ_2 and $\chi'_2 \leq \chi_2$. But here $\chi = \chi_2 = \frownie$ so $\chi'_2 = \frownie$. As $G_b \vdash b$, x_2 must be defined after x in b, so it must be in $\mathbf{dom}(b_3)$.
4. $x_1, x_2 \in \mathbf{dom}(b_2)$. If all the nodes are in $\mathbf{dom}(b_2)$, then $\gamma \subset \rightarrow_{b_2}$ directly. Otherwise, the first node not in $\mathbf{dom}(b_2)$ in γ is necessarily x. Let x_3 be the node just before it. γ has to continue after x, because it has to go back to a node in $\mathbf{dom}(b_2)$, by hypothesis. Let x_4 be the node just after the first occurence of x. γ has the shape $\gamma_1^{\chi_1}, (x_3, x, \chi_3), (x, x_4, \chi_4), \gamma_2^{\chi_2}$.

   - If γ_2 is empty, then as $\mathbf{Degree}(x, b_2(x_4)) = \chi_4$, by lemma 5 there exists an edge $x \xrightarrow{\chi'_4}_b x$, with $\chi'_4 \leq \chi_4$. But here $\chi_4 = \chi$, so we are in the second case and $x \xrightarrow{\chi'_4}_b{}^{+} x$ with $\chi'_4 \leq \chi$.
   - Otherwise, by induction hypothesis on $(x, x_4, \chi_4), \gamma_2^{\chi_2}$, we obtain a path ${\gamma'_2}^{\chi'_2} \subset \rightarrow_b$, from x to x and $\chi'_2 \leq \chi_2$, which means that $x \xrightarrow{\chi'_2}_b{}^{+} x$, and that is enough.

□

Corollary 4 (Correct internal merge)
If $b = (b_1, x = \mathbf{let\ rec}\ b_2\ \mathbf{in}\ e, b_3), \vdash b, \vdash b_2$, $\mathbf{dom}(b_2) \perp \mathbf{dom}(b) \cup \mathbf{FV}(b_1, b_3)$, *and* $b' = b_1, b_2, x = e, b_3$, *then* $\vdash b'$.

Proof We want to prove that if $x_1 \xrightarrow{\frownie}_{b'}{}^{+} x_2$, then $x_1 >_{b'} x_2$ (x_1 is defined before x_2 in b').

- If $x_1, x_2 \in \mathbf{dom}(b)$, by lemma 6, there is a path $x_1 \xrightarrow{\frownie}_b{}^{+} x_2$, and as $\vdash b$, $x_1 >_b x_2$, so $x_1 >_{b'} x_2$.
- If $x_1 \in \mathbf{dom}(b), x_2 \in \mathbf{dom}(b_2)$, by lemma 6, there is a path $x_1 \xrightarrow{\frownie}_b{}^{+} x$, so $x_1 >_b x$ and therefore $x_1 >_{b'} x_2$.
- If $x_1 \in \mathbf{dom}(b_2), x_2 \in \mathbf{dom}(b)$, by lemma 6, then $x_2 \in \{x\} \cup \mathbf{dom}(b_3)$, so $x_1 >_{b'} x_2$.
- If $x_1, x_2 \in \mathbf{dom}(b_2)$, by lemma 6, we are in one of the following two cases.
  - There exists a path $x_1 \xrightarrow{\frownie}_{b_2}{}^{+} x_2$, and as $\vdash b_2$, $x_1 >_{b_2} x_2$, so $x_1 >_{b'} x_2$.
  - There exists a path $x \xrightarrow{\frownie}_b{}^{+} x$, which is impossible, since $\vdash b$.

□

There is a similar property for merging two externally nested bindings – i.e. the second one appears right under the first one.

Lemma 5 (Correct external merge)
If $\mathbf{dom}(b_2) \perp (\mathbf{dom}(b_1) \cup \mathbf{FV}(b_1))$, $\vdash b$ *and* $\vdash b_2$, *then with* $b = b_1, b_2 \vdash b$.

Proof Let $\gamma^{\frownie}$ be a path of $\rightarrow_b$. We prove that it goes from left to right in b.

- If it is a path of $\rightarrow_{b_1}$, then by hypothesis, it goes from left to right.
- If it is a path of $\rightarrow_{b_2}$, then by hypothesis it goes from left to right.
- If it goes from a node defined in b_1 to a node defined in b_2, ok, it goes from left to right.
- It cannot go from node defined in b_2 to a node defined in b_1, because $\mathbf{dom}(b_2) \perp \mathbf{FV}(b_1)$.

□

4.3 Soundness

We first state the two traditional type well-formedness and weakening lemmas.

Proposition 7 (Types well-formed) *If the types in* Γ *are well-formed, and* $\Gamma \vdash e : T$*, then* T *is well-formed.*

Proof By induction on the typing derivation.

Struct. $e = \langle \iota \mid o \rangle$ and $T = \langle I; O; D \rangle$. By syntactic correctness, $\mathbf{dom}(\iota) \perp \mathbf{Names}(o)$, so $\mathbf{dom}(I) \perp \mathbf{dom}(O)$. Moreover, the targets of D, by construction of $\rightarrow_{\langle \iota | o \rangle}$, and $\rightarrow^{\square}_{\langle \iota | o \rangle}$, are in $\mathbf{dom}(O)$, and by typing $\vdash G_{\langle \iota | o \rangle}$, so $\vdash D$. Eventually, $\vdash I$ is given by the typing rule, and $\vdash O$ is obtained by induction hypothesis.

Sum. Assume $e = e_1 + e_2$, $\Gamma \vdash e_1 : \langle I_1; O_1; D_1 \rangle$, $\Gamma \vdash e_2 : \langle I_2; O_2; D_2 \rangle$, $\vdash D_1 \cup D_2$, and $T = \langle (I_1 \cup I_2) \setminus (O_1 \uplus O_2); (O_1 \uplus O_2); D_1 \cup D_2 \rangle$. By induction hypothesis the types of e_1 and e_2 are well-formed, so $I_1 \cup I_2$ and $O_1 \uplus O_2$ are as well. By construction, the inputs are disjoint from the outputs, the graph is correct, and its targets are in $\mathbf{dom}(O_1 \uplus O_2)$.

Freeze. $e = e' \,!\, X$, $\Gamma \vdash e' : \langle I; O; D \rangle$, and $T = \langle I; O; D \,!\, X \rangle$. The only difficulty is to show that the targets of $D \,!\, X$ are in $\mathbf{dom}(O)$, but the ones of D are by induction hypothesis, so it is the same for the ones of $D!X \triangleright x$, and therefore for the ones of D.

Close. Simple by induction hypothesis.

Project and delete. Easy by induction hypothesis. For projection for example, everything is trivial, except maybe that $\mathbf{Targets}(D_{|\mathcal{N}}) \subset \mathbf{dom}(O_{|\mathcal{N}})$, but by induction hypothesis $\mathbf{Targets}(D) \subset \mathbf{dom}(O)$, and as $\mathbf{Targets}(D_{|\mathcal{N}}) = \mathbf{Targets}(D) \cap \mathcal{N}$, we have $\mathbf{Targets}(D_{|\mathcal{N}}) \subset \mathbf{dom}(O) \cap \mathcal{N} = \mathbf{dom}(O_{|\mathcal{N}})$.

Show and hide. Assume $\Gamma \vdash e : \langle I; O; D \rangle$, and by rule (T-Show), $\Gamma \vdash e_{:X_1 \ldots X_n} : \langle I; O_{|X_1 \ldots X_n}; D_{:X_1 \ldots X_n} \rangle$. By induction hypothesis, $\langle I; O; D \rangle$ is well-formed, so $\vdash I$, $\vdash O$, $\vdash D$, $\mathbf{dom}(I) \perp \mathbf{dom}(O)$, and $\mathbf{Targets}(D) \subset \mathbf{dom}(O)$. We can deduce that $\langle I; O_{|X_1 \ldots X_n}; D_{:X_1 \ldots X_n} \rangle$ is well-formed, since $\mathbf{Targets}(D_{:X_1 \ldots X_n}) \subset \{X_1 \ldots X_n\}$ and by typing $\{X_1 \ldots X_n\} \subset \mathbf{dom}(O)$. The other conditions are easy, and hide is similar.

Rename. $e = e'[r]$, $\Gamma \vdash e' : \langle I; O; D \rangle$, $(\mathbf{cod}(r) \setminus \mathbf{dom}(r)) \perp \mathbf{dom}(I) \cup \mathbf{dom}(O)$ (1), and $T = \langle I\{r\}; O\{r\}; D\{r\} \rangle$.

By induction hypothesis, $\mathbf{dom}(I) \perp \mathbf{dom}(O)$, $\mathbf{Targets}(D) \subset \mathbf{dom}(O)$, $\vdash I$, $\vdash O$ and $\vdash D$. Furthermore, $I\{r\}$ and $O\{r\}$ are well-defined individually, but it is not trivial that they do not define the same name twice.

To show this, first remark that $\mathbf{dom}(I\{r\}) = (\mathbf{dom}(I) \setminus \mathbf{dom}(r)) \uplus r(\mathbf{dom}(I))$ and $\mathbf{dom}(O\{r\}) = (\mathbf{dom}(O) \setminus \mathbf{dom}(r)) \uplus r(\mathbf{dom}(O))$.

But by induction hypothesis, we know that $\mathbf{dom}(I) \perp \mathbf{dom}(O)$, so

$$\begin{aligned} \mathbf{dom}(I\{r\}) \cap \mathbf{dom}(O\{r\}) \subset \quad & ((\mathbf{dom}(I) \setminus \mathbf{dom}(r)) \cap \mathbf{cod}(r)) \\ & \cup ((\mathbf{dom}(O) \setminus \mathbf{dom}(r)) \cap \mathbf{cod}(r)) \\ & \cup (r(\mathbf{dom}(I)) \cap r(\mathbf{dom}(O))). \end{aligned}$$

But by (1), both $(\mathbf{dom}(I) \setminus \mathbf{dom}(r)) \cap \mathbf{cod}(r)$ and $(\mathbf{dom}(O) \setminus \mathbf{dom}(r)) \cap \mathbf{cod}(r)$ are empty. Finally, as r is injective and $\mathbf{dom}(I) \perp \mathbf{dom}(O)$, we have $r(\mathbf{dom}(I)) \perp r(\mathbf{dom}(O))$.

Moreover, by induction hypothesis, $\mathbf{Targets}(D) \subset \mathbf{dom}(O)$, so $\mathbf{Targets}(D\{r\}) \subset r(\mathbf{Targets}(D)) \cup (\mathbf{Targets}(D) \setminus \mathbf{dom}(r))$. But $\mathbf{Targets}(D) \setminus \mathbf{dom}(r) \subset (\mathbf{dom}(O) \setminus \mathbf{dom}(r))$, so

$$\mathbf{Targets}(D\{r\}) \subset r(\mathbf{dom}(O)) \cup (\mathbf{dom}(O) \setminus \mathbf{dom}(r)) = \mathbf{dom}(O\{r\}).$$

Split. Assume $\Gamma \vdash e : \langle I; O; D \rangle$, and by rule (T-Split), $\Gamma \vdash e_{X \succ Y} : \langle I \uplus \{X : O(X)\}; O\{X \mapsto Y\}; D_{X \succ Y} \rangle$. By induction hypothesis, $\mathbf{Targets}(D) \subset \mathbf{dom}(O)$. But $\mathbf{Targets}(D_{X \succ Y}) = \mathbf{Targets}(D) \setminus \{X\} \cup \{Y\} \subset \mathbf{dom}(O)\{X \mapsto Y\}$. The other conditions are easy.

Other cases. Easy.

□

Lemma 6 (Weakening) *If* $\Gamma \vdash e : T$ *and* $\mathbf{dom}(\Gamma') \perp \mathbf{FV}(e)$, *then* $\Gamma\langle\Gamma'\rangle \vdash e : T$.

Proof Simple induction on the typing derivation. Clashes of $\mathbf{dom}(\Gamma')$ with bound variables of e' are not a problem because in the rules, new bindings override previous ones. □

Now, typing is preserved by the computational contraction rules.

Lemma 7 (Subject contraction) *If* $e \rightsquigarrow_c e'$ *and* $\Gamma \vdash e : T$, *then* $\Gamma \vdash e' : T$.

Proof By case analysis on the contraction step.

- (Sum). Assume $e = \langle \iota_1 \mid o_1 \rangle + \langle \iota_2 \mid o_2 \rangle$, and $\Gamma \vdash e : T$. By typing we have $\Gamma \vdash \langle \iota_1 \mid o_2 \rangle : \langle I_1; O_1; D_1 \rangle$, $\Gamma \vdash \langle \iota_2 \mid o_2 \rangle : \langle I_2; O_2; D_2 \rangle$, and $T = \langle I; O; D \rangle$, with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = O_1 \uplus O_2$, $D = D_1 \cup D_2$, and $\vdash D$. We have $e' = \langle \iota \mid o \rangle$, where $\iota = (\iota_1 \cup \iota_2) \setminus \mathbf{Input}(o_1, o_2)$, $o = o_1 \succ_\iota o_2$, with $\langle \iota_1 \mid o_1 \rangle \square \langle \iota_2 \mid o_2 \rangle$.

  By lemma 1, $\vdash \rightarrow_{e'}$ and $D = \lfloor \rightarrow_{e'} \rfloor$.

  Then we deduce easily that $\Gamma \vdash e' : T$:

  - $\mathbf{dom}(\iota) = \mathbf{dom}(I)$ is trivial.
  - We have seen that $\vdash \rightarrow_{\langle \iota \mid o \rangle}$.
  - By typing there exist correct Γ_1 and Γ_2 such that $\Gamma\langle I_1 \circ \iota_1^{-1} \uplus \Gamma_1 \rangle \vdash o_1 : \Gamma_1$ and $\Gamma\langle I_2 \circ \iota_2^{-1} \uplus \Gamma_2 \rangle \vdash o_2 : \Gamma_2$. So it would be enough to derive $\Gamma' \vdash o : (\Gamma_1 \uplus \Gamma_2)$, where $\Gamma' = \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_1 \uplus \Gamma_2 \rangle$.

    First $\mathbf{Variables}(o_1) \perp \mathbf{Variables}(o_2)$, so $\mathbf{dom}(\Gamma_1) \perp \mathbf{dom}(\Gamma_2)$.

    Then, $I = I_1' \uplus I_2'$, with $I_1' = I_1 \setminus O_2$ and $I_2' = I_2 \setminus (I_1 \cup O_1)$ and we obtain $I \circ \iota^{-1} = (I_1' \circ \iota^{-1}) \uplus (I_2' \circ \iota^{-1}) = (I_1' \circ \iota_1^{-1}) \uplus (I_2' \circ \iota_2^{-1})$.

    Moreover, with $P = \mathbf{cod}(\iota_1) \cap \mathbf{Variables}(o_2)$, we have $\Gamma_2 = \Gamma_{2\mid P} \uplus \Gamma_{2 \setminus P}$, and by $\langle \iota_1 \mid o_1 \rangle \square \langle \iota_2 \mid o_2 \rangle$, for all $x \in P$, there is a name $X \in \mathbf{dom}(\iota_1) \cap \mathbf{Names}(o)$ such that $(X \triangleright x) \in \iota_1 \cap \mathbf{Input}(o_2)$, so $id_{\mid P} = \mathbf{Input}(o_2) \circ \iota_1^{-1}$, and therefore

$$\begin{array}{rcl} \Gamma_{2\mid P} & = & \Gamma_2 \circ (id_{\mid P}) \\ & = & \Gamma_2 \circ \mathbf{Input}(o_2) \circ \iota_1^{-1} \\ & = & (\Gamma_2 \circ \mathbf{Input}(o_2)) \circ \iota_1^{-1} \\ & = & O_2 \circ \iota_1^{-1} \\ & = & O_{2\mid \mathbf{dom}(\iota_1)} \circ \iota_1^{-1} \\ & = & (O_2 \cap I_1) \circ \iota_1^{-1}. \end{array}$$

    So, we obtain $\Gamma_2 = (O_2 \cap I_1) \circ \iota_1^{-1} \uplus \Gamma_{2 \setminus P}$ and so

$$\begin{array}{rcl} \Gamma' & = & \Gamma\langle \Gamma_1 \uplus ((I_1 \setminus O_2) \circ \iota_1^{-1}) \uplus (I_2' \circ \iota_2^{-1}) \uplus (I_1 \cap O_2) \circ \iota_1^{-1} \uplus \Gamma_{2 \setminus P} \rangle \\ & = & \Gamma\langle \Gamma_1 \uplus I_1 \circ \iota_1^{-1} \uplus (I_2' \circ \iota_2^{-1}) \uplus \Gamma_{2 \setminus P} \rangle. \end{array}$$

    By compatibility, this weakening does not concern free variables of o_1, so we obtain by lemma 6: $\Gamma' \vdash o_1 : \Gamma_1$, and by symmetry $\Gamma' \vdash o_2 : \Gamma_2$, so $\Gamma' \vdash o : \Gamma_1 \uplus \Gamma_2$.

- (Lift). Let $e = \mathbb{L}[\mathbf{let\ rec}\ b\ \mathbf{in}\ e_1]$, and $\Gamma \vdash e : T$, $\mathbf{dom}(b) \perp \mathbf{FV}(\mathbb{L})$, and $e' = \mathbf{let\ rec}\ b\ \mathbf{in}\ \mathbb{L}[e_1]$. By case on $\mathbb{L}$. For example $\mathbb{L} = \square + e_2$, we have a derivation of the shape

$$\dfrac{\dfrac{\vdots}{\vdash \Gamma_b} \quad \dfrac{\vdots}{\vdash b} \quad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash b : \Gamma_b} \quad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash e_1 : T_1}}{\Gamma \vdash \mathbf{let\ rec}\ b\ \mathbf{in}\ e_1 : T_1} \quad \dfrac{\vdots}{\Gamma \vdash e_2 : T_2} \quad \begin{array}{l} I_1 \uplus O_1 \square I_2 \uplus O_2 \\ \vdash D_1 \cup D_2 \end{array}}{\Gamma \vdash (\mathbf{let\ rec}\ b\ \mathbf{in}\ e_1) + e_2 : T}$$

  where $T_1 = \langle I_1; O_1; D_1 \rangle$, $T_2 = \langle I_2; O_2; D_2 \rangle$,
  and $T = \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \uplus O_2; D_1 \cup D_2 \rangle$.

  By hypothesis, $\mathbf{dom}(\Gamma_b) = \mathbf{dom}(b) \perp \mathbf{FV}(e_2)$, so by lemma 6, we have $\Gamma\langle\Gamma_b\rangle \vdash e_2 : T_2$, and we can reconstruct the derivation as follows:

$$\dfrac{\begin{array}{lcccc} \vdash \Gamma_b & \vdots & I_1 \uplus O_1 \sqcap I_2 \uplus O_2 & \vdots & \vdots \\ \vdash b & \Gamma\langle\Gamma_b\rangle \vdash b : \Gamma_b & \vdash D_1 \cup D_2 & \Gamma\langle\Gamma_b\rangle \vdash e_1 : T_1 & \Gamma\langle\Gamma_b\rangle \vdash e_2 : T_2 \\ & & \multicolumn{3}{c}{\Gamma\langle\Gamma_b\rangle \vdash e_1 + e_2 : T} \end{array}}{\Gamma \vdash \textbf{let rec } b \textbf{ in } e_1 + e_2 : T}$$

- (FREEZE). Assume $e = \langle \iota \mid o \rangle ! X$ and $e' = \langle \iota \mid o' \rangle$, with $o = (o_1, X[y^*] \triangleright x = f, o_2)$, and $o' = (o_1, \_[y^*] \triangleright x = f, o_2, X \triangleright y = x)$, with a fresh y.

  By typing, we have a derivation of the shape

$$\dfrac{\dfrac{\begin{array}{l} \vdash I \quad \vdash \Gamma_o \\ \textbf{dom}(\iota) = \textbf{dom}(I) \quad \vdash \rightarrow_{\langle\iota|o\rangle} \end{array} \qquad \dfrac{\forall z \in \textbf{Variables}(o), \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o(z) : \Gamma_o(z)}{\Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}}{\Gamma \vdash \langle \iota \mid o \rangle : \langle I; O; D \rangle} \quad X \in \textbf{dom}(O)}{\Gamma \vdash e : \langle I; O; \lfloor D \,!\, X \rfloor \rangle}$$

  with $O = \Gamma_o \circ \textbf{Input}(o)$ and $D = \lfloor \rightarrow_{\langle\iota|o\rangle} \rfloor$.

  Let $\Gamma_{o'} = \Gamma_o\langle y \mapsto \Gamma_o(x)\rangle$. By weakening, we can derive

$$\forall z \in \textbf{Variables}(o') \setminus \{x, y\}, \Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash o'(z) : \Gamma'_o(z).$$

  For x and y, we easily derive too that

$$\Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash f : \Gamma'_o(x)$$
$$\Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash x : \Gamma'_o(y).$$

  So we have

$$\forall z \in \textbf{Variables}(o'), \Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash o'(z) : \Gamma'_o(z).$$

  Moreover, by lemma 1, we have $\vdash \rightarrow_{e'}$ and $\lfloor \rightarrow_{e'} \rfloor = \lfloor D \,!\, X \rfloor$, so we can derive

$$\dfrac{\dfrac{\begin{array}{l} \vdash I \quad \vdash \Gamma'_o \\ \textbf{dom}(\iota) = \textbf{dom}(I) \quad \vdash \rightarrow_{e'} \end{array} \qquad \dfrac{\forall z \in \textbf{Variables}(o'), \Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash o'(z) : \Gamma'_o(z)}{\Gamma\langle I \circ \iota^{-1} \uplus \Gamma'_o\rangle \vdash o' : \Gamma'_o}}{\Gamma \vdash \langle \iota \mid o' \rangle : \langle I; O; \lfloor D \,!\, X \rfloor \rangle}$$

- (DELETE). Let $e = \langle \iota \mid o \rangle_{|-X_1 \ldots X_n}$, with $\mathcal{N} = \{X_1 \ldots X_n\}$, we have

$$\dfrac{\dfrac{\textbf{dom}(\iota) = \textbf{dom}(I) \quad \vdash I \quad \vdash \Gamma_o \quad \vdash \rightarrow_{\langle\iota|o\rangle} \quad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \iota \mid o \rangle : \langle I; O; D \rangle}}{\Gamma \vdash e : T}$$

  with $T = \langle I'; O'; D' \rangle = \langle I \uplus O_{|\mathcal{N}}; O_{\setminus\mathcal{N}}; D_{|-\mathcal{N}} \rangle$ and $D = \lfloor \rightarrow_{\langle\iota|o\rangle} \rfloor$. But necessarily, we have $e' = \langle \iota' \mid o' \rangle = \langle \iota, \textbf{Input}(o)_{|\mathcal{N}} \mid o_{\setminus\mathcal{N}} \rangle$.

  So, $I \circ \iota^{-1} \uplus \Gamma_o = \big((I \uplus O_{|\mathcal{N}}) \circ (\iota, \textbf{Input}(o)_{|\mathcal{N}})^{-1}\big) \uplus \Gamma'_o$, with $\Gamma'_o = \Gamma_{o\setminus\mathcal{N}}$, and so $\Gamma\langle\big((I \uplus O_{|\mathcal{N}}) \circ (\iota, \textbf{Input}(o)_{|\mathcal{N}})^{-1}\big) \uplus \Gamma'_o\rangle \vdash o_{\setminus\mathcal{N}} : \Gamma'_o$.

  Moreover, we have by lemma 1, $\vdash \rightarrow_{e'}$ and $D' = \lfloor \rightarrow_{e'} \rfloor$, so we can derive

$$\dfrac{\dfrac{\textbf{dom}(\iota') = \textbf{dom}(I') \quad \vdash I' \quad \vdash \Gamma'_o \quad \vdash \rightarrow_{\langle\iota'|o'\rangle} \quad \Gamma\langle I' \circ \iota'^{-1} \uplus \Gamma'_o\rangle \vdash o' : \Gamma'_o}{\Gamma \vdash e' : \langle I'; O'; D' \rangle}}{}$$

- (PROJECT). Let $e = \langle \iota \mid o \rangle_{|X_1 \dots X_n}$. Let $\mathcal{N} = \{X_1 \dots X_n\}$, $\mathcal{N}' = \mathbf{Names}(o) \setminus \mathcal{N}$, and $e'' = \langle \iota \mid o \rangle_{|-\mathcal{N}'}$. We have in fact that $e'' \rightsquigarrow_c e'$, because of the duality of delete and project.

  So if we show that $\Gamma \vdash e'' : T$, we can reproduce exactly the delete case as above.

  By typing, we have $\Gamma \vdash \langle \iota \mid o \rangle : \langle I; O; D \rangle$, and $T = \langle I'; O'; D' \rangle$, with $I' = I \cup O_{\setminus \mathcal{N}}$, $O' = O_{|\mathcal{N}}$, and $D' = D_{|\mathcal{N}}$.

  But we can derive $\Gamma \vdash e'' : \langle I''; O''; D'' \rangle$, with $I'' = I \cup O_{|\mathcal{N}'} = I \cup O_{\setminus \mathcal{N}} = I'$, $O'' = O_{\setminus \mathcal{N}'} = O_{|\mathcal{N}} = O'$, and $D'' = D_{|-\mathcal{N}'} = D_{|\mathcal{N}} = D'$, so we derive $\Gamma \vdash e'' : T$, and may apply the same process as above to deduce $\Gamma \vdash e' : T$.

- (RENAME). Let $e = \langle \iota \mid o \rangle[r]$. We have $e' = \langle \iota\{r\} \mid o\{r\} \rangle$, and by typing:

$$\dfrac{\dfrac{\vdash \to_{\langle \iota | o \rangle} \qquad \dfrac{\vdash I \qquad \vdash \Gamma_o}{\mathbf{dom}(I) = \mathbf{dom}(\iota)} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \iota \mid o \rangle : \langle I; O; D \rangle} \qquad (\mathbf{cod}(r) \setminus \mathbf{dom}(r)) \perp (\mathbf{dom}(I) \cup \mathbf{dom}(O))}{\Gamma \vdash e : T}$$

  with $T = \langle I'; O'; D' \rangle = \langle I\{r\}; O\{r\}; D\{r\} \rangle$, $D = \lfloor \to_{\langle \iota | o \rangle} \rfloor$,
  and $O = \Gamma_o \circ \mathbf{Input}(o)$.

  We may write e' as $\langle \iota' \mid o' \rangle = \langle \iota\{r\} \mid o\{r\} \rangle$, and $\mathbf{Input}(o') = \mathbf{Input}(o) \circ r^{-1}_{\mathbf{Names}(o)}$, so

$$\begin{aligned} \Gamma_o \circ \mathbf{Input}(o') &= \Gamma_o \circ \mathbf{Input}(o) \circ r^{-1}_{\mathbf{Names}(o)} \\ &= O \circ r^{-1}_{\mathbf{Names}(o)} \\ &= O \circ r^{-1}_{\mathbf{dom}(O)} \\ &= O'. \end{aligned}$$

  For inputs, we have $I' \circ \iota'^{-1} = I\{r\} \circ (\iota\{r\})^{-1} = I \circ r^{-1}_{\mathbf{dom}(I)} \circ r_{\mathbf{dom}(\iota)} \circ \iota^{-1} = I \circ \iota^{-1}$, so $\Gamma\langle I' \circ \iota'^{-1} \uplus \Gamma_o \rangle \vdash o' : \Gamma_o$.

  Moreover, it is easily seen that $\mathbf{dom}(I') = \mathbf{dom}(\iota')$, $\vdash I'$, and by lemma 1, we have, with $\vdash \to_{\langle \iota' | o' \rangle}$ and $D' = \lfloor \to_{\langle \iota' | o' \rangle} \rfloor$, so we can derive

$$\dfrac{\vdash \to_{\langle \iota' | o' \rangle} \qquad \dfrac{\vdash I' \qquad \vdash \Gamma_o}{\mathbf{dom}(\iota') = \mathbf{dom}(I')} \qquad \Gamma\langle I' \circ \iota'^{-1} \uplus \Gamma_o \rangle \vdash o' : \Gamma_o}{\Gamma \vdash \langle \iota' \mid o' \rangle : \langle I'; O'; D' \rangle}$$

- (CLOSE). Let $e = \nabla\langle \epsilon \mid o \rangle$. We have $e' = \mathbf{let\ rec}\ \ \mathbf{Bind}(\overline{o})\ \ \mathbf{in}\ \ \mathbf{Record}(o)$, and $\vdash \mathbf{Bind}(\overline{o})$, and by typing

$$\dfrac{\dfrac{\vdash \Gamma_o \qquad \to_{\langle \epsilon | o \rangle} \qquad \Gamma\langle \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \epsilon \mid o \rangle : \langle \emptyset; O; D \rangle}}{\Gamma \vdash e : T}$$

  with $T = \{O\}$, $D = \lfloor \to_{\langle \epsilon | o \rangle} \rfloor$, and $O = \Gamma_o \circ \mathbf{Input}(o)$.

  Let

$$\begin{aligned} &o = d_1 \dots d_n \\ &d_i = L_i[x^i_1 \dots x^i_{n_i}] \triangleright x_i = e_i \\ &b = \mathbf{Bind}(o) = (x_1 = e_1 \dots x_n = e_n) \\ &s = \mathbf{Record}(o) = (X_1 = x_{\mu(1)} \dots X_m = x_{\mu(m)}) \\ &\text{where } \mu : \{1 \dots m\} \to \{1 \dots n\} \text{ injective} \\ &\text{and for all } i, X_i = L_{\mu(i)}. \end{aligned}$$

  We have $e' = \mathbf{let\ rec}\ \ b\ \ \mathbf{in}\ \ \{s\}$ and let $\Gamma_o = \{x_i : T_i \mid i \in \{1 \dots n\}\}$.

  We have

- $\Gamma\langle\Gamma_o\rangle \vdash b : \Gamma_o$ (easy with $\Gamma\langle\Gamma_o\rangle \vdash o : \Gamma_o$),
- $\vdash b$, by lemma 4,
- $\Gamma\langle\Gamma_o\rangle \vdash \{s\} : \{O\}$, since for all $i \in \{1 \dots m\}$, $\Gamma\langle\Gamma_o\rangle \vdash x_{\mu(i)} : \Gamma_o(x_{\mu(i)})$, and $\Gamma_o(x_{\mu(i)}) = O(X_i)$,

so it is ok.

- (SHOW). Assume $e = e_{0:X_1 \dots X_n}$, with $e_0 = \langle \iota \mid o \rangle$. Then, $e' = \langle \iota \mid o' \rangle$, and $o' = \mathbf{Show}(o, X_1 \dots X_n)$. Let $\mathcal{N} = \{X_1 \dots X_n\}$. The typing derivation is of the shape

$$\dfrac{\dfrac{\vdash \rightarrow_{\langle \iota \mid o \rangle} \qquad \dfrac{\vdash I \qquad \vdash \Gamma_o}{\mathbf{dom}(I) = \mathbf{dom}(\iota)} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash e_0 : \langle I; O; D\rangle} \qquad \mathcal{N} \subset \mathbf{dom}(O)}{\Gamma \vdash e : \langle I; O_{\mid \mathcal{N}}; D'\rangle}$$

with $D = \lfloor \rightarrow_{\langle \iota \mid o \rangle} \rfloor$, $D' = \lfloor D_{:\mathcal{N}} \rfloor$, and $O = \Gamma_o \circ \mathbf{Input}(o)$.

By lemma 1, we have $\vdash \rightarrow_{e'}$ and $D' = \lfloor G_{e'} \rfloor$.

The typing of o' is exactly as the one for o, so we obtain that e' has type $\langle I; O'; D'\rangle$, with $O' = \Gamma_o \circ \mathbf{Input}(o')$. But $\mathbf{Input}(o') = \mathbf{Input}(o)_{\mid \mathcal{N}}$, so $O' = O_{\mid \mathcal{N}}$, which is the expected result.

- (HIDE). As for delete and project, we obtain the expected result by reasoning dually to the (SHOW) case.
- (SPLIT). Let $e_0 = \langle \iota \mid o \rangle$ and $e = e_{0\, X \succ Y}$, with $o = (o_1, X[z^*] \triangleright x = e_1, o_2)$. We have $e' = \langle \iota' \mid o' \rangle = \langle \iota, X \triangleright x \mid o_1, X[z^*] \triangleright y = e_1, o_2 \rangle$ for a fesh y.

  The typing derivation is of the shape

$$\dfrac{\dfrac{\vdash \rightarrow_{\langle \iota \mid o \rangle} \qquad \dfrac{\vdash I \qquad \vdash \Gamma_o}{\mathbf{dom}(I) = \mathbf{dom}(\iota)} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash e_0 : \langle I; O; D\rangle} \qquad Y \notin \mathbf{dom}(O) \cup \mathbf{dom}(I)}{\Gamma \vdash e_{X \succ Y} : \langle I \uplus \{X : O(X)\}; O\{X \mapsto Y\}; D_{X \succ Y}\rangle}$$

  with $D = \lfloor \rightarrow_{\langle \iota \mid o \rangle} \rfloor$, and $O = \Gamma_o \circ \mathbf{Input}(o)$.

  By lemma 1, we have $\vdash \rightarrow_{\langle \iota' \mid o' \rangle}$ and $\lfloor \rightarrow_{\langle \iota' \mid o' \rangle} \rfloor = D_{X \succ Y}$.

  Moreover, the environment $\Gamma_{o'}$ corresponding to o' is $\Gamma_o\{x \mapsto y\}$, and it is easy to reconstruct the derivation for e' (by a weakening).

□

It is now possible to prove that if a well-typed expression reduces to another expression, then this expression has the same type, which is known as the subject reduction property.

First we prove that typing is compositional at the level of lift contexts.

Lemma 8 (Lift context) *If* $\Gamma \vdash \mathbb{L}[e] : T$, $\Gamma \vdash e : T'$, *and* $\Gamma \vdash e' : T'$, *then* $\Gamma \vdash \mathbb{L}[e'] : T$.

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

- $\mathbb{L} = \{\mathbb{S}\}$, with $\mathbb{S} = s_v, X = \square, s$. We have a derivation of the form

$$\dfrac{\dfrac{\vdots}{\forall (Y = f) \in (s_v, s), \Gamma \vdash f : O(Y)} \qquad \dfrac{\vdots}{\Gamma \vdash e : T'}}{\Gamma \vdash \{\mathbb{S}[e]\} : T}$$

  with $T = \{O \cup \{X : T'\}\}$.

By hypothesis we have $\Gamma \vdash e' : T'$, so we can reconstruct the derivation

$$\frac{\dfrac{\vdots}{\forall (Y = f) \in (s_v, s), \Gamma \vdash f : O(Y)} \qquad \dfrac{\vdots}{\Gamma \vdash e' : T'}}{\Gamma \vdash \{\mathbb{S}\,[e]\} : T}$$

- $\mathbb{L} = op[\Box]$, for $op \in \{\nabla, [r], !X, _{|-X_1 \ldots X_n}, _{|X_1 \ldots X_n}\}$. We have a derivation of the shape

$$\frac{\Gamma \vdash e : T' \qquad \text{side conditions}}{\Gamma \vdash e : op[T']}$$

  with $T = op[T']$, and op deduced from the typing rules. The only side conditions appearing in the rules are $X \in \mathbf{dom}(O)$ for freezing and $\mathbf{cod}(r) \perp \mathbf{dom}(I) \cup \mathbf{dom}(O)$ for renaming, which do not use the shape of e, so we can reconstruct the derivation in a compositional way.

- $\mathbb{L} = \Box + e_1$. The derivation is of the form

$$\frac{I_1 \uplus O_1 \,\Box\, I_2 \uplus O_2 \qquad \vdash D_1 \cup D_2 \qquad \Gamma \vdash e : \langle I_1; O_1; D_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2; O_2; D_2 \rangle}{\Gamma \vdash e + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \uplus O_2; D_1 \cup D_2 \rangle}$$

  Similarly, we can reconstruct the derivation compositionally with e'.

- $\mathbb{L} = v + \Box$. Similar.

□

This property is true for multiple lift contexts as well.

Lemma 9 (Multiple lift context) *If* $\Gamma \vdash \mathbb{F}\,[e] : T$, $\Gamma \vdash e : T'$, *and* $\Gamma \vdash e' : T'$, *then* $\Gamma \vdash \mathbb{F}\,[e'] : T$.

Proof By trivial induction on $\mathbb{F}$. □

Corollary 5 (External substitution) *If* $\Gamma \vdash v : \Gamma(x)$, *and* $\Gamma \vdash \mathbb{F}\,[x] : T$, *then* $\Gamma \vdash \mathbb{F}\,[v] : T$.

Proof Trivial. □

For evaluation contexts, typing is not exactly compositional, since in the **let rec** case, it depends on the shapes of the bindings. However, we have this slightly less general property.

Lemma 10 (Evaluation context) *Assume* $\Gamma \vdash \mathbb{E}\,[e] : T$, *with a sub-derivation* $\Gamma\langle\Gamma'\rangle \vdash e : T'$ *in place of the hole. Assume also that* $\Gamma\langle\Gamma'\rangle \vdash e' : T'$, *that* $e \in$ **Predictable** *and* $e' \in$ **Predictable**, *and that for all* $x \in \mathbf{FV}(e')$, $x \in \mathbf{FV}(e)$ *and* $\mathbf{Degree}(x, e) \leq \mathbf{Degree}(x, e')$.

Then $\Gamma \vdash \mathbb{E}\,[e'] : T$.

Proof By induction on $\mathbb{E}$.

- $\mathbb{E} = \mathbb{F}$. By lemma 9.

- $\mathbb{E} =$ **let rec** b_v **in** $\mathbb{F}$. The derivation has shape

$$\frac{\vdash b_v \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_{b_v}\rangle \vdash b_v : \Gamma_{b_v}} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_{b_v}\rangle \vdash \mathbb{F}\,[e] : T}}{\Gamma \vdash \textbf{let rec } b_v \textbf{ in } \mathbb{F}\,[e] : T}$$

  By lemma 9, we have $\Gamma \vdash \mathbb{F}\,[e'] : T$, so we can reconstruct the derivation compositionally.

- $\mathbb{E} = \textbf{let rec } \mathbb{B}[\mathbb{F}[e]] \textbf{ in } f$, with $\mathbb{B} = b_v, x = \square, b$. The derivation has the shape

$$\dfrac{\dfrac{\dfrac{\vdots}{\forall y \neq x \in \textbf{dom}(\mathbb{B}), \Gamma\langle\Gamma_b\rangle \vdash \mathbb{B}(y) : \Gamma_b(y)} \quad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash \mathbb{F}[e] : \Gamma_b(x)}}{\Gamma\langle\Gamma_b\rangle \vdash \mathbb{B}[\mathbb{F}[e]] : \Gamma_b} \quad \dfrac{\dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash f : T} \quad \vdash b}{}}{\Gamma \vdash \mathbb{E}[e] : T}$$

  where $b = \mathbb{B}[\mathbb{F}[e]]$.

  By induction hypothesis, we derive $\Gamma\langle\Gamma_b\rangle \vdash \mathbb{F}[e'] : \Gamma_b(x)$.

  Let $b' = \mathbb{B}[\mathbb{F}[e']]$. There only remains to prove that $\vdash b'$.

  As $\vdash b$, we have $\succcurlyeq_b \vdash b'$, since they define the same variables in the same order.

  Obviously, we have $>_b = >_{b'}$.

  By hypothesis and hypothesis 1, we have $\mathbb{F}[e] \in \textbf{Predictable}$ iff $\mathbb{F}[e'] \in \textbf{Predictable}$, so b and b' are equivalent with respect to shapes.

  For dependencies, we know that the edges with a target different from x in $\rightarrow_b$ stay the same in $\rightarrow_{b'}$. For the edges towards x, we know that $\textbf{FV}(\mathbb{F}[e']) \subset \textbf{FV}(\mathbb{F}[e])$. Let $y \in \textbf{FV}(\mathbb{F}[e']) \cap \textbf{dom}(\mathbb{B})$. By hypothesis and hypothesis 2, we have $\textbf{Degree}(y, \mathbb{F}[e]) \leq \textbf{Degree}(y, \mathbb{F}[e'])$, so that if $y \xrightarrow{\odot}{}^{+}_{b'} z$, then $y \xrightarrow{\odot}{}^{+}_{b} z$. Therefore, the constraints imposed on the ordering are weaker than in b, and by lemma 3, the order of definition stays acceptable.

□

Lemma 11 (Evaluation context) *If $e \rightsquigarrow_c e'$, and $\Gamma \vdash \mathbb{E}[e] : T$, then $\Gamma \vdash \mathbb{E}[e'] : T$.*

Proof By lemma 10. □

Now that we have proven that typing is preserved by the (Context) rule, the last difficulty for proving subject reduction concerns the (Subst) rule. Indeed, replacing a variable with its value might change the shape of a binding. We first prove that if the variable is defined above the current context, it does not change the typing.

Now, we check that substituting a variable with its value, defined in the current binding does not change typing either.

Lemma 12 (Internal substitution preserves correct ordering) *Let $\mathbb{B} = (b_v, y = \square, b_1)$, $b = \mathbb{B}[\mathbb{F}[\mathbb{N}[x]]]$, $b' = \mathbb{B}[\mathbb{F}[\mathbb{N}[v]]]$, $b_v(x) = v$, and $\textbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}]) \perp \textbf{FV}(v) \cup \{x\}$. If $\vdash b$, then $\vdash b'$.*

Proof Assume $\vdash b$. Then, b and b' define the same variables in the same order. So, $\succcurlyeq_b \vdash b'$.

By hypothesis 1, if $\mathbb{F}[\mathbb{N}[x]] \in \textbf{Predictable}$, then $\mathbb{F}[\mathbb{N}[v]] \in \textbf{Predictable}$, so the shapes of b' are less restrictive than in b.

For this, by lemma 3, it is enough to show that $\rightarrow_b < \rightarrow_{b'}$.

For this we remark that

$$\rightarrow_{b'} \subset \rightarrow_b \cup \{z \xrightarrow{\chi} y \mid z \in \textbf{FV}(\mathbb{F}[\mathbb{N}[v]]), \chi = \textbf{Degree}(z, \mathbb{F}[\mathbb{N}[v]])\}$$

But by hypothesis 2, among the variables $z \in \textbf{FV}(\mathbb{F}[\mathbb{N}[v]])$, we can distinguish two cases.

- For variables $z \in \textbf{FV}(v) \setminus \textbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])$, we have $\chi = \odot$.
- For variables $z \notin \textbf{FV}(v) \setminus \textbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])$, we have $\chi = \textbf{Degree}(z, \mathbb{F}[\mathbb{N}])$.

Therefore, we have

$$\begin{aligned} \rightarrow_{b'} \subset \quad & \rightarrow_b \\ & \cup\{z \xrightarrow{\odot} y \mid z \in \textbf{FV}(v) \setminus \textbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])\} \\ & \cup\{z \xrightarrow{\chi} y \mid z \in \textbf{FV}(\mathbb{F}[\mathbb{N}[v]]) \cap (\textbf{FV}(v) \setminus \textbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])), \chi = \textbf{Degree}(z, \mathbb{F}[\mathbb{N}])\} \end{aligned}$$

Let $\rightarrow''$ be the right member of the above equation.

For each edge in $\{z \xrightarrow{\odot} y \mid z \in \mathbf{FV}(v) \setminus \mathbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])\}$, as $z \in \mathbf{FV}(v)$, there is an edge $z \xrightarrow{\chi}_b x$. But by hypothesis 2, $\mathbf{Degree}(x, \mathbb{E}[\mathbb{N}[x]]) = \odot$, so there is a strict path from z to y in $\rightarrow_b$.

For each edge in $\{z \xrightarrow{\chi} y \mid z \in \mathbf{FV}(\mathbb{F}[\mathbb{N}[v]]) \cap (\mathbf{FV}(v) \setminus \mathbf{Capt}_{\square}(\mathbb{F}[\mathbb{N}])), \chi = \mathbf{Degree}(z, \mathbb{F}[\mathbb{N}])\}$, we have $\mathbf{Degree}(z, \mathbb{F}[\mathbb{N}[x]]) \leq \chi$. (This can be deduced from hypothesis 2.)

So, we have $\rightarrow_b < \rightarrow''$, and by transitivity of graph comparison, we get $\rightarrow_b < \rightarrow_{b'}$.

□

We can eventually verify that reduction through the (SUBST) rule preserves types.

Lemma 13 (Access) *If* $\mathbb{E}[\mathbb{N}](x) = v$ *and* $\Gamma \vdash \mathbb{E}[\mathbb{N}[x]] : T$, *then* $\Gamma \vdash \mathbb{E}[\mathbb{N}[v]] : T$.

Proof By induction on $\mathbb{E}$.

- $\mathbb{E} = \mathbb{F}$, impossible.
- $\mathbb{E} = \textbf{let rec } b_v \textbf{ in } \mathbb{F}$. By corollary 5.
- $\mathbb{E} = \textbf{let rec } \mathbb{B}[\mathbb{F}] \textbf{ in } e$. Let $b = \mathbb{B}[\mathbb{F}[\mathbb{N}[x]]]$, $b' = \mathbb{B}[\mathbb{F}[\mathbb{N}[v]]]$, and $\mathbb{B} = b_v, y = \square, b_1$.

  The derivation has the shape

$$\dfrac{\vdash b \qquad \dfrac{\vdots}{\forall y \in \mathbf{dom}(b_v, b_1), \Gamma\langle\Gamma_b\rangle \vdash \mathbb{B}(y) : \Gamma(y)} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash \mathbb{F}[\mathbb{N}[x]] : \Gamma(x)}}{\Gamma \vdash \mathbb{E}[\mathbb{N}[x]] : T}$$

  We have $b_v(x) = v$, and by lemma 12, $\vdash b'$.

  Eventually, we have $\Gamma\langle\Gamma_b\rangle \vdash v : \Gamma(x)$, so by corollary 5, we can derive $\Gamma\langle\Gamma_b\rangle \vdash \mathbb{F}[\mathbb{N}[v]] : \Gamma(x)$, and therefore

$$\dfrac{\vdash b' \qquad \dfrac{\vdots}{\forall y \in \mathbf{dom}(b'), \Gamma\langle\Gamma_b\rangle \vdash b' : \Gamma_b(y)}}{\Gamma \vdash \mathbb{E}[\mathbb{N}[v]] : T}$$

□

Type preservation along the (IM) rule is proven.

Lemma 14 (Internal merge) *If* $e = \textbf{let rec } b_v, x = (\textbf{let rec } b_1 \textbf{ in } e_1), b_2 \textbf{ in } f$ $e' = \textbf{let rec } b_v, b_1, x = e_1, b_2 \textbf{ in } f$, *and* $\Gamma \vdash e : T$, *then* $\Gamma \vdash e' : T$.

Proof

We have a derivation of the shape

$$\dfrac{\dfrac{\forall y \neq x \quad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash b(y) : \Gamma_b(y)}}{\Gamma\langle\Gamma_b\rangle \vdash b : \Gamma_b} \qquad \dfrac{\dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash f : T} \qquad \begin{matrix}\vdash \Gamma_b \\ \vdash b\end{matrix}}{\Gamma \vdash \textbf{let rec } b_v, x = (\textbf{let rec } b_1 \textbf{ in } e_1), b_2 \textbf{ in } f : T}}{\Gamma\langle\Gamma_b\rangle \vdash \textbf{let rec } b_1 \textbf{ in } e_1 : \Gamma_b(x)} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash e_1 : \Gamma_b(x)}}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash b_1 : \Gamma_{b_1}} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash e_1 : \Gamma_b(x)}}{\Gamma\langle\Gamma_b\rangle \vdash \textbf{let rec } b_1 \textbf{ in } e_1 : \Gamma_b(x)}$$

where $b = b_v, x = (\textbf{let rec } b_1 \textbf{ in } e_1), b_2$.

Let $b' = b_v, b_1, x = e_1, b_2$. By corollary 4, we have $\vdash b'$.

Moreover, by weakening, we have

$$\forall y \neq x \quad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash b(y) : \Gamma_b(y)}$$

and with $\Gamma_{b'} = \Gamma_b \uplus \Gamma_{b_1}$,

$$\frac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle : f : T}$$

and we have

$$\frac{\forall y \in \mathbf{dom}(b') \; \dfrac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle \vdash b'(y) : \Gamma_{b'}(y)} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle : f : T} \qquad \begin{array}{l}\vdash \Gamma_{b'} \\ \vdash b'\end{array}}{\Gamma \vdash \mathbf{let\ rec}\ b'\ \mathbf{in}\ f : T}$$

□

Next, rule (EM) is examined.

Lemma 15 (External merge)
If $\mathbf{dom}(b) \perp (\mathbf{dom}(b_v) \cup \mathbf{FV}(b_v))$, *then* $\Gamma \vdash e'_0 : T$.
$e_0 = \mathbf{let\ rec}\ b_v\ \mathbf{in}\ \mathbf{let\ rec}\ b\ \mathbf{in}\ e$,
$e'_0 = \mathbf{let\ rec}\ b_v, b\ \mathbf{in}\ e$, *and*
$\Gamma \vdash e_0 : T$,

Proof The typing derivation for e_0 has the shape

$$\frac{\vdash b_v \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle \vdash b_v : \Gamma_1} \qquad \dfrac{\vdash b \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle\langle\Gamma_2\rangle \vdash b : \Gamma_2} \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle\langle\Gamma_2\rangle \vdash e : \Gamma_2}}{\Gamma\langle\Gamma_1\rangle \vdash \mathbf{let\ rec}\ b\ \mathbf{in}\ e : \Gamma_1}}{\Gamma \vdash e_0 : T}$$

By lemma 5, we have $\vdash b_v, b$.
So by weakening we can reconstruct the derivation.
□
We can now state the subject reduction property.

Lemma 16 (Subject reduction) *If* $e \longrightarrow e'$ *and* $\Gamma \vdash e : T$, *then* $\Gamma \vdash e' : T$.

Proof By immediate induction, with lemmas 7, 11, 13, and 15. □
Eventually, we prove that if a term is well-typed and is not a result, then either it reduces to another term, or it is stuck on a free variable. This is known as the progress property.

Lemma 17 (Progress) *If* $\Gamma \vdash e : T$ *and* e *is not a result, then either* $e = \mathbb{E}[\mathbb{N}[x]]$ *with* $x \notin \mathbf{Capt}_{\square}(\mathbb{E}[\mathbb{N}])$, *or there exists* e' *such that* $e \longrightarrow e'$.

Proof By induction on e.

1. If e is of the shape $\mathbb{L}[e_0]$, and e_0 is not a value. If $e_0 = \mathbf{let\ rec}\ b\ \mathbf{in}\ f$, then the (Lift) applies. Else, by induction hypothesis we are in one of the following cases.
   - $e_0 = \mathbb{E}[\mathbb{N}[x]]$ with $x \notin \mathbf{Capt}_{\square}(\mathbb{E}[\mathbb{N}])$, and e is stuck on x too, i.e. $e = \mathbb{L}[\mathbb{E}[\mathbb{N}[x]]]$.
   - Otherwise, if $e_0 \longrightarrow e'_0$, we reason by case analysis on the applied reduction rule.
     - (EM). Then the (Lift) rule applies for e.
     - (Subst) or (Context). Then $e_0 = \mathbb{E}[f]$ and $e'_0 = \mathbb{E}[f']$. By case analysis again, on $\mathbb{E}$:
       * If $\mathbb{E} = \square$ or $\mathbb{E} = \mathbb{F}$, then e reduces by the same rule, since $\mathbb{L}[\mathbb{E}]$ is an evaluation context.
       * If $\mathbb{E} = \mathbf{let\ rec}\ b_v\ \mathbf{in}\ \mathbb{F}_0$ or $\mathbb{E} = \mathbf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathbf{in}\ g$, then the (Lift) rule applies for e.
2. If e is of the shape $\mathbb{N}[x]$, there is nothing to show (x is necessarily free in $\mathbb{N}[x]$).
3. $e = \mathbf{let\ rec}\ b\ \mathbf{in}\ f$.

(a) Else, if b is evaluated. $b = b_v$. If f is a result, it has the shape **let rec** $b_v{}'$ **in** v (or e would be one), and rule (EM) applies.

Otherwise, by induction hypothesis, we are in one of the two following cases.

- $f \longrightarrow f'$. By case analysis on the reduction:
  - (EM). Then rule (EM) applies for e as well.
  - (Subst) or (Context).
    We have $f = \mathbb{E}[g]$ and $f' = \mathbb{E}[g']$. If $\mathbb{E} =$ **let rec** $b_v{}'$ **in** $\mathbb{F}'$ or $\mathbb{E} =$ **let rec** $\mathbb{B}[\mathbb{F}]$ **in** g, then rule (EM) applies, and otherwise, the same rule applies for e since **let rec** b_v **in** $\mathbb{E}$ is an evaluation context.
- $f = \mathbb{E}[\mathbb{N}[x]]$, with $x \notin \mathbf{Capt}_\Box(\mathbb{E}[\mathbb{N}])$.
  If $\mathbb{E} =$ **let rec** $b_v{}'$ **in** $\mathbb{F}$ or $\mathbb{E} =$ **let rec** $\mathbb{B}[\mathbb{F}]$ **in** g, then rule (EM) applies, and otherwise, $\mathbb{E}$ is of the shape $\mathbb{F}$ and $f = \mathbb{F}[\mathbb{N}[x]]$, $e =$ **let rec** b_v **in** $\mathbb{F}[\mathbb{N}[x]]$. If $x \in \mathbf{dom}(b_v)$, then rule (Subst) applies, and otherwise $e = \mathbb{E}_0[\mathbb{N}[x]]$ with $x \notin \mathbf{Capt}_\Box(\mathbb{E}_0)$.

(b) Otherwise, b is not evaluated, so b is of the shape $b_v, y = g, b_1$, where g is not a value.

- If g is a result, then it is of the shape **let rec** $b_v{}'$ **in** v, and by internal merge, $e \longrightarrow$ **let rec** $b_v, b_v{}', y = v, b_1$ **in** f.
- Otherwise, by induction hypothesis:
  - If $g \longrightarrow g'$, by case on the reduction.
    - (EM): then rule (IM) applies for e.
    - (Context) or (Subst): then $g = \mathbb{E}[g_0]$ and $g' = \mathbb{E}[g_0']$. If $\mathbb{E}$ is of the shape **let rec** $b_v{}'$ **in** $\mathbb{F}$ or **let rec** $\mathbb{B}[\mathbb{F}]$ **in** g'', then rule (IM) applies for e, and otherwise, the global context is an evaluation context and the same rule (Context) or (Subst) applies for e.
  - If $g = \mathbb{E}[\mathbb{N}[x]]$ with $x \notin \mathbf{Capt}_\Box(\mathbb{E}[\mathbb{N}])$. By case on $\mathbb{E}$. First notice that we know that $x \notin \mathbf{dom}(y = g, b_1)$, since by typing $\vdash b$ and therefore if $x \xrightarrow{\text{☹}}{}^{+}_{b} y$, then x is defined before y in b, and $g = \mathbb{E}[\mathbb{N}[x]]$ implies the existence of an edge $x \xrightarrow{\text{☹}}_b y$ by hypothesis 2.
    - If $\mathbb{E} =$ **let rec** $b_v{}'$ **in** $\mathbb{F}$ or **let rec** $\mathbb{B}[\mathbb{F}]$ **in** g'', then rule (IM) applies.
    - Else, if $x \in \mathbf{dom}(b_v)$, then rule (Subst) applies, since the global context is an evaluation context.
    - Else, if $x \notin \mathbf{dom}(b_v, y = g, b_1)$, then e is of the shape $\mathbb{E}_0[\mathbb{N}[x]]$ with $x \notin \mathbf{Capt}_\Box(\mathbb{E}_0)$.

4. $e = e_1 + e_2$.

We treated the case where either e_1 or e_2 is not a value above. So we may assume that both are values. The typing derivation must be of the shape

$$\dfrac{\dfrac{\vdash \rightarrow_{\langle \iota_1 | o_1 \rangle} \quad \vdash I_1 \quad \vdash \Gamma_1 \qquad \dfrac{\vdots}{\Gamma\langle I_1 \circ \iota_1^{-1} \uplus \Gamma_1\rangle \vdash o_1 : \Gamma_1}}{\mathbf{dom}(\iota_1) = \mathbf{dom}(I_1) \qquad \Gamma \vdash e_1 : \langle I_1; O_1; D_1\rangle} \qquad \dfrac{\vdash \rightarrow_{\langle \iota_2 | o_2 \rangle} \quad \vdash I_2 \quad \vdash \Gamma_2 \qquad \dfrac{\vdots}{\Gamma\langle I_2 \circ \iota_2^{-1} \uplus \Gamma_2\rangle \vdash o_2 : \Gamma_2}}{\mathbf{dom}(\iota_2) = \mathbf{dom}(I_2) \qquad \Gamma \vdash e_2 : \langle I_2; O_2; D_2\rangle}}{\Gamma \vdash e : T}$$

with $\vdash D_1 \cup D_2$ and $I_1 \uplus O_1 \,\square\, I_2 \uplus O_2$ as side-conditions and

$$\begin{array}{lll} D_1 = \rightarrow_{\langle \iota_1 | o_1 \rangle} & & T = \langle I; O; D\rangle \\ D_2 = \rightarrow_{\langle \iota_2 | o_2 \rangle} & \text{and} & I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \\ O_1 = \Gamma_1 \circ \mathbf{Input}(o_1) & & O = O_1 \uplus O_2 \\ O_2 = \Gamma_2 \circ \mathbf{Input}(o_2) & & D = D_1 \cup D_2. \end{array}$$

But values with mixin types may only be of two kinds: either variables or structures. If either one of the two is a variable, we have treated the case as well in the beginning (and $e = \mathbb{E}[\mathbb{N}[x]]$ with $x \notin \mathbf{Capt}_\Box(\mathbb{E})$).

So we may assume that $e_1 = \langle \iota_1 \mid o_1 \rangle$, $e_2 = \langle \iota_2 \mid o_2 \rangle$, and that bound variables of the two structures meet only on the common names, i.e. $e_1 \,\square\, e_2$. This can be reached via α-conversion.

Moreover, typing imposes that $\mathbf{Names}(O_1) \perp \mathbf{Names}(O_2)$, so $\mathbf{Names}(o_1) \perp \mathbf{Names}(o_2)$, and rule (Sum) applies.

5. (CLOSE). $e = \nabla e_0$, and e_0 is a value, not a variable (these cases have been treated above). By typing, $e_0 = \langle \epsilon \mid o \rangle$ and we have

$$\dfrac{\dfrac{\vdash \Gamma_o \qquad \vdash \rightarrow_{\langle \epsilon | o \rangle} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_o\rangle \vdash o : \Gamma_o}}{\Gamma \vdash e_0 : \langle \emptyset; O; D \rangle}}{\Gamma \vdash e : \{O\}}$$

So we have $e \longrightarrow$ **let rec** $\mathbf{Bind}(\overline{o})$ **in** $\mathbf{Record}(o)$, provided $\overline{o}$ is defined and $\mathbf{Bind}(\overline{o})$ is syntactically correct.

By lemma 2, $\overline{o}$ is defined and $\vdash \mathbf{Bind}(\overline{o})$.

For any forward reference from x to y in $\mathbf{Bind}(\overline{o})$, there is an edge from y to x in $\rightarrow_o$, and if it points to a component of unpredictable shape, then either its degree is $\circledcirc$ or we have $y \succ_o x$, so y is defined before x in $\overline{o}$ and therefore, in both cases, x is defined before y in $\mathbf{Bind}(\overline{o})$.

6. Other operators trivial.

□

Eventually, we can prove a standard soundness theorem.

Theorem 1 (Soundness) *The evaluation of a well-typed expression may either not terminate, or reach a result, or get stuck on a free variable.*

5 Conclusion

We have presented a language of call-by-value mixin modules, equipped with a reduction semantics and a sound type system. A companion paper, in preparation, formalizes the compilation of the **let rec** construct of **MM**.

Some open issues remain to be dealt with, which are related to different practical uses of mixin modules. If mixin modules are used as first-class, core language constructs, then the simple type system presented here is not expressive enough. Some form of polymorphism over mixin signatures seems necessary, along the lines of type systems for record concatenation proposed by Harper and Pierce [13] and by Pottier [18]. If one wants to build a module system based on mixin modules, then type abstraction and user-defined type components have to be considered. We are working on an extension of Leroy's module system [15] to mixin modules with type components.

Furthermore, in both cases, the programmer will probably have to write interfaces for mixin modules. In the type system presented here, the type of a mixin module must include its dependency graph. This is problematic for two reasons: first, it is cumbersome to write the whole graph of a mixin module; second, the dependency graph reveals too much information on the implementation of the mixin module, in the sense that small changes in the implementation must be reflected in the dependency graph, thus changing the interface. An issue that remains open is how to reduce the amount of dependency information that needs to be put in mixin interfaces, while still allowing static checking of well-formedness.

References

[1] Davide Ancona. *Modular Formal Frameworks for Module Systems.* PhD thesis, Universita di Pisa, 1998.

[2] Davide Ancona and Elena Zucca. A calculus of module systems. *J. Func. Progr.*, 2002.

[3] Gérard Boudol. The recursive record semantics of objects revisited. In David Sands, editor, *Europ. Symp. on Progr. 2001*, volume 2028 of *LNCS*, pages 269–283. Springer-Verlag, 2001.

[4] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA90*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, 1990.

[6] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.

[7] Luca Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang.*, pages 266–277. ACM Press, 1997.

[8] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Int. Conf. on Functional Progr. 96*, pages 262–273. ACM Press, 1996.

[9] Levent Erkök, John Launchbury, and Andrew Moran. Semantics of fixIO. Fixed Points in Comp. Sc. 2001.

[10] Matthew Flatt. *Programming Languages for Reusable Software Components.* PhD thesis, Rice University, 1999.

[11] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl. 1998*, pages 236–248. ACM Press, 1998.

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

[13] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *symp. Principles of Progr. Lang. 1991*, pages 131–142, Orlando, Florida, 1991.

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

[15] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symp. Principles of Progr. Lang.*, pages 109–122. ACM Press, 1994.

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

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised).* The MIT Press, 1997.

[18] Francois Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[19] Claudio V. Russo. *Types for Modules.* PhD thesis, University of Edinburgh, 1998.

[20] João Costa Seco and Luís Caires. A basic model of typed components. In *Europ. Conf. on Object-Oriented Progr. 2000*, volume 1850, pages 108–128, 2000.

[21] Joe B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*, pages 412–428. Springer-Verlag, 2000.

[22] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. and Comp.*, 1992.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399