Mixin modules in a call-by-value setting

Tom Hirschowitz and Xavier Leroy

INRIA Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
Fax: +33 1 39 63 56 84
`{Tom.Hirschowitz,Xavier.Leroy}@inria.fr`

Abstract. The ML module system provides powerful parameterization facilities, but lacks the ability to split mutually recursive definitions across modules, and does not provide enough facilities for incremental programming. A promising approach to solve these issues is Ancona and Zucca's mixin modules calculus *CMS*. However, the straightforward way to adapt it to ML fails, because it allows arbitrary recursive definitions to appear at any time, which ML does not support. In this paper, we enrich *CMS* with a refined type system that controls recursive definitions through the use of dependency graphs. We then develop and prove sound a separate compilation scheme, directed by dependency graphs, that translate mixin modules down to a CBV λ-calculus extended with a non-standard `let rec` construct.

1 Introduction

Modular programming and code reuse is easier if the programming language provides adequate features to support them. Three important such features are (1) *parameterization*, which allows reusing a module in different contexts; (2) *overriding and late binding*, which supports incremental programming by refinements of existing modules; and (3) *cross-module recursion*, which allows definitions to be spread across several modules, even if they mutually refer to each other. Many programming languages provide two of these features, but not all three: class-based object-oriented languages provide (2) and (3), but are weak on parameterization (1); conventional linkers, as well as linking calculi [8], have cross-module recursion built in, and sometimes provide facilities for overriding, but lack parameterization; finally, ML functors and Ada generics provide powerful parameterization mechanisms, but prohibit cross-module recursion and offer no direct support for late binding.

The concept of *mixins*, first introduced as a generalization of inheritance in class-based OO languages [7], then extended to a family of module systems [12, 3, 14, 19], offers a promising and elegant solution to this problem. A mixin is a collection of named components, either defined (bound to a definition) or deferred (declared without definition). The basic operation on mixins is the sum, which takes two mixins and connects the defined components of one with the similarly-named deferred components of the other; this provides natural support for cross-mixin recursion. A mixin is named and can be summed several times with different mixins; this allows powerful parameterization, including but not restricted to an encoding of ML functors. Finally, the mixin calculus of Ancona and Zucca [3] supports both late binding and early binding of defined components, along with deleting and renaming operations, thus providing excellent support for incremental programming.

Our long-term goal is to extend the ML module system with mixins, taking Ancona and Zucca's *CMS* calculus [3] as a starting point. There are two main issues: one, which we leave for future work, is to support type components in mixins; the other, which we address in this paper, is to equip *CMS* with a call-by-value semantics consistent with that of the core ML language. Shifting *CMS* from its original call-by-name semantics to a call-by-value semantics requires a precise control of recursive definitions created by mixin composition. The call-by-name semantics of *CMS* puts no restrictions on recursive definitions, allowing ill-founded ones such as `let rec x = 2 * y and y = x + 1`, causing the program to diverge when `x` or `y` is selected. In an ML-like, call-by-value setting, recursive definitions are statically restricted to syntactic values, *e.g.* `let rec f =` λ`x...` `and g =` λ`y...` This provides stronger guarantees (ill-founded recursions are detected at compile-time rather than at run-time), and supports more efficient compilation of recursive definitions. Extending these two desirable properties to mixin modules in the presence of separate compilation [8, 16] is challenging: illegal recursive definitions can appear a posteriori when we take the sum `A + B` of two mixin modules, at a time where only the signatures of `A` and `B` are known, but not their implementations.

The solution we develop here is to enrich the *CMS* type system, adding graphs in mixin signatures to represent the dependencies between the components. The resulting typed calculus, called CMS_v, guarantees that recursive definitions created by mixin composition evaluate correctly under a call-by-value regime, yet leaves considerable flexibility in composing mixins. We then provide a type-directed, separate compilation scheme for CMS_v. The target of this compositional translation is λ_B, a simple call-by-value λ-calculus with a non-standard `let rec` construct in the style of Boudol [5]. Finally, we prove that the compilation of a type-correct CMS_v mixin is well typed in a sound, non-standard type system for λ_B that generalizes that of [5], thus establishing the soundness of our approach.

The remainder of the paper is organized as follows. Section 2 gives a high-level overview of the *CMS* and CMS_v mixin calculi, and explains the recursion problem. Section 3 defines the syntax and typing rules for CMS_v, our call-by-value mixin module calculus. The compilation scheme (from CMS_v to λ_B) is presented in section 4. In section 5, we equip λ_B with a type system guaranteeing the proper call-by-value evaluation of recursive definitions, and use it to show the correctness of the compilation scheme. We review related work in section 6, and conclude in section 7.

2 Overview

2.1 The *CMS* calculus of mixins

We start this paper by an overview of the *CMS* module calculus of [3], using an ML-like syntax for readability. A basic mixin module is similar to an ML structure, but may contain "holes":

```
mixin Even =  mix
  ? val odd: int -> bool             (* odd is deferred *)
  let even = λx. x = 0 or odd(x-1)   (* even is defined  *)
end
```

In other terms, a mixin module consists of defined components, `let`-bound to an expression, and deferred components, declared but not yet defined. The fundamental operator on mixin modules is the sum, which combines the components of two mixins, connecting defined and deferred components having the same names. For example, if we define `Odd` as

```
mixin Odd = mix
  ? val even: int -> bool
  let odd = λx. x = 1 or even(x-1)
end
```

the result of `mixin Nat = Even + Odd` is equivalent to writing

```
mixin Nat = mix
  let even = λx. x = 0 or odd(x-1)
  let odd  = λx. x = 1 or even(x-1)
end
```

As in class-based languages, all defined components of a mixin are mutually recursive by default; thus, the above should be read as the ML structure

```
module Nat = struct
  let rec even = λx. x = 0 or odd(x-1)
      and odd  = λx. x = 1 or even(x-1)
end
```

Another commonality with classes is that defined components are late bound by default: the definition of a component can be overridden later, and other definitions that refer to this component will "see" the new definition. The overriding is achieved in two steps: first, deleting the component via the \ operator, then redefining it via a sum. For instance,

```
mixin Nat' = (Nat \ even) + (mixin let even = λx. x mod 2 = 0 end)
```

is equivalent to the direct definition

```
mixin Nat’ = mix
  let even = λx. x mod 2 = 0
  let odd  = λx. x = 1 or even(x-1)
end
```

Early binding (definite binding of a defined name to an expression in all other components that refer to this name) can be achieved via the freeze operator !. For instance, `Nat ! odd` is equivalent to

```
mix
  let even = let odd = λx. x = 1 or even(x-1) in
             λx. x = 0 or odd(x-1)
  let odd  = λx. x = 1 or even(x-1)
end
```

For convenience, our CMS_v calculus also provides a `close` operator that freezes all components of a mixin in one step. Projections (extracting the value of a mixin component) are restricted to closed mixins – for the same reasons that in class-based languages, one cannot invoke a method directly from a class: an instance of the class must first be taken using the "new" operator.

A component of a mixin can itself be a mixin module. Not only does this provide ML-style nested mixins, but it also supports a general encoding of ML functors [2]. Consider the following ML functor definition and applications.

```
module F = functor (X : S) -> struct ... end
module R = F(A)
module S = F(B)
```

We can achieve the same effect in CMS_v by representing `F` as a mixin with a deferred mixin component representing its formal parameter, then summing it twice with the actual arguments `A` and `B`.

```
mixin F = mix
  ? mixin Arg : S
  mixin X = Arg
  mixin Res = mix ... end
end
mixin R = close(F + mix mixin Arg = A end).Res
mixin S = close(F + mix mixin Arg = B end).Res
```

2.2 Controlling recursive definitions

It is well known that general recursive definitions, whose right-hand sides involve arbitrary computation, require call-by-name or call-by-need (lazy) evaluation, via on-demand unfolding. If the recursive definition is not well founded, as in `let rec x = y + 1 and y = 2 * x`, the program will diverge the first time the value of `x` or `y` is needed. In contrast, call-by-value evaluation of recursive definitions is usually allowed only if the right-hand sides are syntactic values (*e.g.*

λ-abstractions or constants), thus ruling out the example above. In return, the programmer obtains the guarantee that the recursive definition is well-founded, evaluates in one step, and will not cause divergence nor re-computation when the recursively-defined identifiers are used.

This semantic issue is exacerbated by mixin modules, which are in essence big mutual `let rec` definitions. Worse, ill-founded recursive definitions such as the above can appear not only when defining a basic mixin such as

```
mixin Bad = close(mix  let x = y + 1   let y = x * 2  end)
```

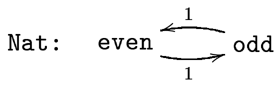
but also *a posteriori* when combining two innocuous-looking mixins:

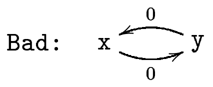
```
mixin OK1 = mix  ? val y : int   let x = y + 1  end
mixin OK2 = mix  ? val x : int   let y = x * 2  end
mixin Bad = close(OK1 + OK2)
```

Although `OK1` and `OK2` contain no ill-founded recursions, the sum `OK1 + OK2` contains one. If the definitions of `OK1` and `OK2` are known when we type-check and compile their sum, we can simply expand `OK1 + OK2` into an equivalent monolithic mixin and reject the faulty recursion. But in a separate compilation setting, `OK1 + OK2` can be compiled in a context where the definitions of `OK1` and `OK2` are not known, but only their signatures. Then, the ill-founded recursion cannot be detected. This is the major problem we face in extending ML with mixin modules.

One partial solution, that we find unsatisfactory, is to rely on lazy evaluation to implement a call-by-name semantics for modules, evaluating components only at selection or when the module is closed. (This is the approach followed by the Moscow ML recursive modules [18], and also by class initialization in Java.) This would have several drawbacks. Besides potential efficiency problems, lazy evaluation does not mix well with ML, which is a call-by-value imperative language. For instance, many ML modules contain side-effecting initialization code that must be evaluated at predictable program points; that would not be the case with lazy evaluation of modules. The second drawback is that ill-founded recursive definitions (as in the `Bad` example above) would not be detected statically, but cause the program to loop or fail at run-time. We believe this seriously decreases program safety: compile-time detection of ill-founded recursive definitions is much preferable.

Our approach consists in enriching mixin signatures with graphs representing the dependencies between components of a mixin module, and rely on these graphs to detect statically ill-founded recursive definitions. For example, the `Nat` and `Bad` mixins shown above have the following dependency graphs:





Edges labeled 0 represent an immediate dependency: the value of the source node is needed to compute that of the target node. Edges labeled 1 represent a delayed dependency, occurring under at least one λ-abstraction; thus, the value of the target node can be computed without knowing that of the source node. Ill-founded recursions manifest themselves as cycles in the dependency graph involving at least one "0" edge. Thus, the correctness criterion for a mixin is, simply: all cycles in its dependency graph must be composed of "1" edges only. Hence, `Nat` is correct, while `Bad` is rejected.

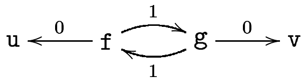
(Notice that the weaker criterion "all cycles contain at least one edge labeled 1" is incorrect, since it would allow ill-founded definitions such as `let rec f = λx. x + y and y = f 0`.)

The power of dependency graphs becomes more apparent when we consider mixins that combine recursive definitions of functions and immediate computations that sit outside the recursion:

```
mixin M1 = mix
  ? val g : ...
  let f = λx. ...g...
  let u = f 0
end
```

```
mixin M2 = mix
  ? val f : ...
  let g = λx. ...f...
  let v = g 1
end
```

The dependency graph for the sum `M1 + M2` is:



It satisfies the correctness criterion, thus accepting this definition; other systems that record a global "valuability" flag on each signature, such as the recursive modules of [10], would reject this definition.

3 The CMS_v calculus

We now define formally the syntax and typing rules of CMS_v, our call-by-value variant of *CMS*.

3.1 Syntax

The syntax of CMS_v terms and types is defined in figure 1. Here, x ranges over a countable set *Vars* of (α-convertible) variables, while X ranges over a countable set *Names* of (non-convertible) names used to identify mixin components.

Although our module system is largely independent of the core language, for the sake of specificity we use a standard simply-typed λ-calculus with constants as core language. Core terms can refer by name to a (core) component of a mixin structure, via the notation $E.X$.

Mixin terms include core terms (proper stratification of the language is enforced by the typing rules), structure expressions building a mixin from a collection of components, and the various mixin operators mentioned in section 2: sum, rename, freeze, delete and close.

Core terms:	$C ::= x \mid cst$	variables, constants
	$\mid \lambda x.C \mid C_1 C_2$	abstraction, application
	$\mid E.X$	component projection
Mixin terms:	$E ::= C$	core term
	$\mid \langle \iota; o \rangle$	mixin structure
	$\mid E_1 + E_2$	sum
	$\mid E[X \leftarrow Y]$	rename X to Y
	$\mid E\,!\,X$	freeze X
	$\mid E \setminus X$	delete X
	$\mid \mathtt{close}(E)$	close
Input assignments:	$\iota ::= x_i \overset{i \in I}{\mapsto} X_i$	ι injective
Output assignments:	$o ::= X_i \overset{i \in I}{\mapsto} E_i$	
Core types:	$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau \to \tau$	
Mixin types:	$\mathcal{T} ::= \tau$	core type
	$\mid \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$	mixin signature
Type assignments:	$\mathcal{I}, \mathcal{O} ::= X_i \overset{i \in I}{\mapsto} \mathcal{T}_i$	

Fig. 1. Syntax of CMS_v

A mixin structure $\langle \iota; o \rangle$ is composed of an *input assignment* ι and an *output assignment* o. The input assignment associates internal variables to names of imported components, while the output assignment associates expressions to names of exported components. These expressions can refer to imported components via their associated internal variables. This explicit distinction between names and internal variables allows internal variables to be renamed by α-conversion, while external names remain immutable, thus making projection by name unambiguous [17, 2, 19].

The notation $x_i \overset{i \in I}{\mapsto} X_i$ denote the unique surjective, finite map ι such that for all $i \in I$, $\iota(x_i) = X_i$. It is valid only if for all $i, j \in I$, if $i \neq j$, then $x_i \neq x_j$. Then $dom(\iota)$ denotes $\{x_i \mid i \in I\}$ and $cod(\iota)$ denotes $\{X_i \mid i \in I\}$. $X_i \overset{i \in I}{\mapsto} E_i$, and $X_i \overset{i \in I}{\mapsto} \mathcal{T}_i$ are defined in the same way.

The notions of free and bound variables, and of substitution are defined in a standard way in figure 2.

As usual, terms are quotiented by a notion of structural equivalence. First, ι, o, and $\mathcal{I}$-$\mathcal{O}$ expressions denoting the same finite map are identified. Furthermore, there is a special notion of α-conversion, formally defined in figure 3. We use the notation $\iota_1 + \iota_2$ for ι_1 and ι_2 such that for all $x \in dom(\iota_1) \cap dom(\iota_2)$, $\iota_1(x) = \iota_2(x)$, denoting the unique finite map ι such that for all $x \in dom(\iota_1)$, $\iota(x) = \iota_1(x)$ and for all $x \in dom(\iota_2)$, $\iota(x) = \iota_2(x)$. The structural equivalence is the least congruence satisfying the two identities in figure 3. The first one is the

Free variables

$$\begin{aligned} FV(x) &= \{x\} & FV(E_1 + E_2) &= FV(E_1) \cup FV(E_2) \\ FV(cst) &= \emptyset & FV(\langle \iota; o \rangle) &= FV(o) \setminus dom(\iota) \\ FV(\lambda x.C) &= FV(C) \setminus \{x\} & FV(X_i \overset{i \in I}{\mapsto} E_i) &= \bigcup_{i \in I} FV(E_i) \\ FV(C_1 C_2) &= FV(C_1) \cup FV(C_2) \end{aligned}$$

$$\begin{aligned} FV(E \,!\, X) &= FV(E.X) = FV(E[X \leftarrow Y]) = FV(E \setminus X) = FV(\mathtt{close}(E)) \\ &= FV(E) \end{aligned}$$

Substitution

Core

$$\begin{aligned} \lambda y.C\{x \leftarrow E\} &= \lambda y.C\{x \leftarrow E\} \ (y \notin FV(E) \cup \{x\}) & x\{x \leftarrow E\} &= E \\ (C_1 C_2)\{x \leftarrow E\} &= C_1\{x \leftarrow E\} C_2\{x \leftarrow E\} & y\{X \leftarrow E\} &= y \quad x \neq y \end{aligned}$$

Mixin

$$\begin{aligned} E'[X \leftarrow Y]\{x \leftarrow E\} &= E'\{x \leftarrow E\}[X \leftarrow Y] & E' \,!\, X\{x \leftarrow E\} &= E'\{x \leftarrow E\} \,!\, X \\ E' \setminus X\{x \leftarrow E\} &= E'\{x \leftarrow E\} \setminus X & E'.X\{x \leftarrow E\} &= E'\{x \leftarrow E\}.X \end{aligned}$$

$$\begin{aligned} (E_1 + E_2)\{x \leftarrow E\} &= E_1\{x \leftarrow E\} + E_2\{x \leftarrow E\} \\ \langle \iota; o \rangle\{x \leftarrow E\} &= \langle \iota; o\{x \leftarrow E\} \rangle \ (x \notin dom(\iota)) \\ (X_i \overset{i \in I}{\mapsto} E_i)\{x \leftarrow E\} &= X_i \overset{i \in I}{\mapsto} E_i\{x \leftarrow E\} \end{aligned}$$

Fig. 2. Functions on terms

standard α-conversion on core terms. The second one is the adapted α-conversion on structures, stating that bound variables can be renamed if no capture occurs.

Due to late binding, a virtual (defined but not frozen) component of a mixin is both imported and exported by the mixin: it is exported with its current definition, but is also imported so that other exported components refer to its final value at the time the component is frozen or the mixin is closed, rather than to its current value. In other terms, component X of $\langle \iota; o \rangle$ is deferred when $X \in cod(\iota) \setminus dom(o)$, virtual when $X \in cod(\iota) \cap dom(o)$, and frozen when $X \in dom(o) \setminus cod(\iota)$.

Example 1 *Consider the following mixin, expressed in the ML-like syntax of section 2.*

```
mix  ? val x: int    let y = x + 2   let z = y + 1   end
```

It is expressed in CMS$_v$ *syntax as the structure* $\langle \iota; o \rangle$, *where* $\iota = [x \mapsto X;\ y \mapsto Y;\ z \mapsto Z]$ *and* $o = [Y \mapsto x + 2;\ Z \mapsto y + 1]$. *The names* X, Y, Z *correspond to the variables in the ML-like syntax, while the variables* x, y, z *bind them locally. Here,* X *is only an input, but* Y *and* Z *are both input and output, since these components are virtual. The definition of* Z *refers to the imported value of* Y, *thus allowing later redefinition of* Y *to affect* Z.

$$\frac{y \notin FV(C)}{\lambda x.C \equiv \lambda y.C\{x \leftarrow y\}} \ \text{(core-alpha)}$$

$$\frac{y \notin FV(o) \cup dom(\iota)}{\langle \iota + \{x \mapsto X\}; o\rangle \equiv \langle \iota + \{y \mapsto X\}; o\{x \leftarrow y\}\rangle} \ \text{(mixin-alpha)}$$

Fig. 3. Structural equivalence

By lack of space, we do not give a source-level dynamic semantics for the CMS_v calculus. (Section 4 gives a semantics by translation into the λ-calculus.)

3.2 Types and dependency graphs

Types $\mathcal{T}$ are either core types (those of the simply-typed λ-calculus) or mixin signatures $\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$. The latter are composed of two mappings $\mathcal{I}$ and $\mathcal{O}$ from names to types, one for input components, the other for output components; and a safe dependency graph $\mathcal{D}$.

A dependency graph $\mathcal{D}$ is a directed multi-graph whose nodes are external names of imported or exported components, and whose edges carry a valuation $\chi \in \{0, 1\}$. An edge $X \xrightarrow{1} Y$ means that the term E defining Y refers to the value of X, but in such a way that it is safe to put E in a recursive definition that simultaneously defines X in terms of Y. An edge $X \xrightarrow{0} Y$ means that the term E defining Y cannot be put in such a recursive definition: the value of X must be entirely computed before E is evaluated. It is generally undecidable whether a dependency is of the 0 or 1 kind, so we take the following conservative approximation: if E is an abstraction $\lambda x.C$, then all dependencies for Y are labeled 1; in all other cases, they are all labeled 0. (Other, more precise approximations are possible, but this one works well enough and is consistent with core ML.)

More formally, for $x \in FV(E)$, we define $\nu(x,\ E) = 1$ if $E = \lambda y.C$ and $\nu(x,\ E) = 0$ otherwise. Given the mixin structure $s = \langle \iota; o\rangle$, we then define its dependency graph $\mathcal{D}(s)$ as follows: its nodes are the names of all components of s, and it contains an edge $X \xrightarrow{\chi} Y$ if and only if there exist E and x such that

$$o(Y) = E \quad \text{and} \quad \iota(x) = X \quad \text{and} \quad x \in FV(E) \quad \text{and} \quad \chi = \nu(x,\ E).$$

We then say that a dependency graph $\mathcal{D}$ is *safe*, and write $\vdash \mathcal{D}$, if all cycles of $\mathcal{D}$ are composed of edges labeled 1. This captures the idea that only dependencies of the "1" kind are allowed inside a mutually recursive definition.

Formally, a path P in $\mathcal{D}$ is defined by

$$\begin{array}{rll} P ::= & [X] & \text{if } X \text{ is mentioned in } \mathcal{D} \\ \mid & (X \xrightarrow{\chi} Y) :: P & \text{if } \mathit{fst}(P) = Y \end{array}$$

where

1. X is mentioned in $\mathcal{D}$ means that $\mathcal{D}$ either contains an edge $X \xrightarrow{\chi} Y$ or an edge $Y \xrightarrow{\chi} X$ or both, for some χ and Y ;
2. and $\mathit{fst}(p)$ and $\mathit{last}(p)$ are defined like this

$$\begin{array}{rcl} \mathit{fst}([X]) = X & \qquad & \mathit{last}([X]) = X \\ \mathit{fst}((X \xrightarrow{\chi} Y) :: P) = X & \qquad & \mathit{last}((X \xrightarrow{\chi} Y) :: P) = \mathit{last}(P) \end{array}$$

Of course, we often will write $X_0 \xrightarrow{\chi_1} X_1 \ldots \xrightarrow{\chi_n} X_n$ for $(X_0 \xrightarrow{\chi_1} X_1) :: \ldots :: (X_{n-1} \xrightarrow{\chi_n} X_n) :: [X_n]$.

A cycle is a path P such that $\mathit{fst}(P) = \mathit{last}(P)$. We define the valuation of a path as the minimum of the valuations of its edges:

$$\begin{array}{c} \nu([X]) = 1 \\ \nu((X \xrightarrow{\chi} Y) :: P) = \min(\chi, \nu(P)) \end{array}$$

Definition 1 *$\vdash \mathcal{D}$ means that for all cycle P of $\mathcal{D}$, $\nu(P) = 1$.*

In order to type-check mixin operators, we must be able to compute the dependency graph for the result of the operator given the dependency graphs for its operands. We now define the graph-level operators corresponding to the mixin operators.

Sum: the sum $\mathcal{D}_1 + \mathcal{D}_2$ of two dependency graphs is simply their union:

$$\mathcal{D}_1 + \mathcal{D}_2 = \{X \xrightarrow{\chi} Y \mid (X \xrightarrow{\chi} Y) \in \mathcal{D}_1 \text{ or } (X \xrightarrow{\chi} Y) \in \mathcal{D}_2\}$$

Rename: assuming Y is not mentioned in $\mathcal{D}$, the graph $\mathcal{D}[X \leftarrow Y]$ is the graph $\mathcal{D}$ where the node X, if any, is renamed Y, keeping all edges unchanged. The notation $X\{Y \leftarrow Z\}$ denotes the straightforward substitution on names.

$$\mathcal{D}[X \leftarrow Y] = \{A\{X \leftarrow Y\} \xrightarrow{\chi} B\{X \leftarrow Y\} \mid (A \xrightarrow{\chi} B) \in \mathcal{D}\}$$

Delete: the graph $\mathcal{D}\backslash X$ is the graph $\mathcal{D}$ where we remove all edges leading to X.

$$\mathcal{D}\backslash X = \mathcal{D} \setminus \{Y \xrightarrow{\chi} X \mid Y \in \mathit{Names}, \chi \in \{0,1\}\}$$

Freeze: operationally, the effect of freezing the component X in a mixin structure is to replace X by its current definition E in all definitions of other exported components. At the dependency level, this causes all components Y that previously depended on X to now depend on the names on which E depends. Thus, paths $Z \xrightarrow{\chi_1} X \xrightarrow{\chi_2} Y$ in the original graph become edges $Z \xrightarrow{\min(\chi_1,\chi_2)} Y$ in the result graph.

$$\begin{array}{rl} & \mathcal{D}!X = (\mathcal{D} \cup \mathcal{D}_{around}) \setminus \mathcal{D}_{remove} \\ \text{where } \mathcal{D}_{around} = & \{Y \xrightarrow{\min(\chi_1,\chi_2)} Z \mid (Y \xrightarrow{\chi_1} X) \in \mathcal{D},\ (X \xrightarrow{\chi_2} Z) \in \mathcal{D}\} \\ \text{and } \quad \mathcal{D}_{remove} = & \{X \xrightarrow{\chi} Y \mid Y \in \mathit{Names},\ \chi \in \{0,1\}\} \end{array}$$

The sum of two safe graphs is not necessarily safe (unsafe cycles may appear); thus, the typing rules explicitly check the safety of the sum. However, it is interesting to note that the other graph operations preserve safety:

$$\Gamma \vdash x : \Gamma(x) \ \text{(var)} \qquad \Gamma \vdash c : TC(c) \ \text{(const)} \qquad \frac{\Gamma + \{x : \tau_1\} \vdash C : \tau_2}{\Gamma \vdash \lambda x.C : \tau_1 \to \tau_2} \ \text{(abstr)}$$

$$\frac{\Gamma \vdash C_1 : \tau' \to \tau \qquad \Gamma \vdash C_2 : \tau'}{\Gamma \vdash C_1 C_2 : \tau} \ \text{(app)} \qquad \frac{\Gamma \vdash E : \{\emptyset; \mathcal{O}; \emptyset\}}{\Gamma \vdash E.X : \mathcal{O}(X)} \ \text{(select)}$$

$$\frac{\begin{array}{c} \vdash \mathcal{D}\langle \iota; o\rangle \qquad \mathit{dom}(o) = \mathit{dom}(\mathcal{O}) \\ \Gamma + \{x : \mathcal{I}(\iota(x)) \mid x \in \mathit{dom}(\iota)\} \vdash o(X) : \mathcal{O}(X) \text{ for } X \in \mathit{dom}(o) \end{array}}{\Gamma \vdash \langle \iota; o\rangle : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\langle \iota; o\rangle\}} \ \text{(struct)}$$

$$\frac{\begin{array}{c} \Gamma \vdash E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\} \qquad \Gamma \vdash E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\} \qquad \vdash \mathcal{D}_1 + \mathcal{D}_2 \\ \mathit{dom}(\mathcal{O}_1) \cap \mathit{dom}(\mathcal{O}_2) = \emptyset \qquad \mathcal{I}_1(X) = \mathcal{I}_2(X) \text{ for all } X \in \mathit{dom}(\mathcal{I}_1) \cap \mathit{dom}(\mathcal{I}_2) \end{array}}{\Gamma \vdash E_1 + E_2 : \{\mathcal{I}_1 + \mathcal{I}_2; \mathcal{O}_1 + \mathcal{O}_2; \mathcal{D}_1 + \mathcal{D}_2\}} \ \text{(sum)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad \mathcal{I}(X) = \mathcal{O}(X)}{\Gamma \vdash E \,!\, X : \{\mathcal{I}_{\setminus X}; \mathcal{O}; \mathcal{D}!X\}} \ \text{(freeze)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad X \in \mathit{dom}(\mathcal{O})}{\Gamma \vdash E \setminus X : \{\mathcal{I}; \mathcal{O}_{\setminus X}; \mathcal{D} \setminus X\}} \ \text{(delete)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad Y \notin \mathit{dom}(\mathcal{I}) \cup \mathit{dom}(\mathcal{O})}{\Gamma \vdash E[X \leftarrow Y] : \{\mathcal{I} \circ [Y \mapsto X]; \mathcal{O} \circ [Y \mapsto X]; \mathcal{D}[X \leftarrow Y]\}} \ \text{(rename)}$$

$$\frac{\Gamma \vdash E : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \qquad \mathit{dom}(\mathcal{I}) \subseteq \mathit{dom}(\mathcal{O}) \qquad \mathcal{I}(X) = \mathcal{O}(X) \text{ for all } X \in \mathit{dom}(\mathcal{I})}{\Gamma \vdash \mathtt{close}(E) : \{\emptyset; \mathcal{O}; \emptyset\}} \ \text{(close)}$$

Fig. 4. Typing rules

Lemma 1 *If $\mathcal{D}$ is a safe dependency graph, then the graphs $\mathcal{D}[X \leftarrow Y]$, $\mathcal{D}\setminus X$ and $\mathcal{D}!X$ are safe.*

The proof is in appendix A.

3.3 Typing rules

The typing rules for CMS_v are shown in figure 4. The typing environment Γ is a finite map from variables to types. We assume given a mapping TC from constants to core types. All dependency graphs appearing in the typing environment and in input signatures are assumed to be safe.

The rules resemble those of [3], with additional manipulations of dependency graphs. Projection of a structure component requires that the structure has no input components. Structure construction type-checks every output component in an environment enriched with the types assigned to the input components; it also checks that the corresponding dependency graph is safe. For the sum operator, both mixins must agree on the types of common input components,

$$\vdash \tau \ \text{(core)}$$

$$\frac{\begin{array}{c} \mathit{Pred}(\mathcal{D}) \subset \mathit{dom}(\mathcal{I}) \qquad \mathit{Succ}(\mathcal{D}) \subset \mathit{dom}(\mathcal{O}) \\ \vdash \mathcal{I}(X) \text{ for all } X \in \mathit{dom}(\mathcal{I}) \qquad \vdash \mathcal{O}(X) \text{ for all } X \in \mathit{dom}(\mathcal{O}) \qquad \vdash \mathcal{D} \end{array}}{\vdash \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}} \ \text{(mixin)}$$

Fig. 5. Well-formed types

and must have no output components in common; again, we need to check that the dependency graph of the sum is safe, to make sure that the sum introduces no illegal recursive definitions. Freezing a component requires that its type in the input signature and in the output signature of the structure are identical, then removes it from the input signature. In contrast, deleting a component removes it from the output signature. Finally, closing a mixin is equivalent to freezing all its input components, and results in an empty input signature and dependency graph.

For simplicity, the rules (sum), (freeze) and (close) require strict syntactic equality of types. Although we will not do it here by lack of space, it is possible to introduce a notion of subtyping [3] corresponding to adding input components, removing output components, and adding "fake" dependencies in dependency graphs.

Our goal is to translate well-typed terms of CMS_v into a simple calculus with `let rec`, relying on the dependency graphs. To do this in a sound way, it is crucial to only have to deal with safe dependency graphs.

For this purpose, we define the notion of a well-formed type, as described in figure 5. A core type is always well-formed, whereas a mixin type is well-formed if all the graphs appearing in it are safe. Our type system satisfies the following well-formedness property.

Lemma 2 *If* $\Gamma \vdash E : \mathcal{T}$ *is derivable, then* $\vdash \mathcal{T}$.

Proof The proof is a simple induction on the proof tree, relying crucially on the condition that all the dependency graphs appearing in the environment and in input signatures are safe, on the lemma 1, and on the safety checks in the rules (sum) and (struct). □

4 Compilation

We now present a compilation scheme translating CMS_v terms into call-by-value λ-calculus extended with records and a `let rec` binding. This compilation scheme is compositional, and type-directed, thus supporting separate compilation.

4.1 Intuitions

A mixin structure is translated into a record, with one field per output component of the structure. Each field corresponds to the expression defining the output component, but λ-abstracts all input components on which it depends, that is, all its predecessors in the dependency graph. These extra parameters account for the late binding semantics of virtual components. Consider again the `M1` and `M2` example at the end of section 2. These two structures are translated to:

```
m1 = { f = λg.λx. ...g...;  u = λf. f 0 }
m2 = { g = λf.λx. ...f...;  v = λg. g 1 }
```

The sum `M = M1 + M2` is then translated into a record that takes the union of the two records `m1` and `m2`:

```
m = { f = m1.f; u = m1.u; g = m2.g; v = m2.v }
```

Later, we close `M`. This requires connecting the formal parameters representing input components with the record fields corresponding to the output components. To do this, we examine the dependency graph of `M`, identifying the strongly connected components and performing a topological sort. We thus see that we must first take a fixpoint over the `f` and `g` components, then compute `u` and `v` sequentially. Thus, we obtain the following code for `close(M)`:

```
let rec f = m.f g and g = m.g f in
let u = m.u f in
let v = m.v g in
{ f = f; g = g; u = u; v = v }
```

Notice that the `let rec` definition we generate is unusual: it involves function applications in the right-hand sides, which is usually not supported in call-by-value λ-calculi. Fortunately, Boudol [5] has already developed a non-standard call-by-value calculus that supports such `let rec` definitions; we adopt a variant of his calculus as our target language.

4.2 The target language

The target language for our translation is the λ_B calculus, a variant of the λ-calculus with records and recursive definitions studied in [5]. Its syntax is as follows:

$$\begin{aligned} M ::= {} & x \mid cst \mid \lambda x.M \mid M_1\ M_2 \\ & \mid \langle X_1 = M_1; \ldots; X_n = M_n \rangle \mid M.X \\ & \mid \textbf{let}\ x = M_1\ \textbf{in}\ M \\ & \mid \textbf{let rec}\ x_1 = M_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = M_n\ \textbf{in}\ M \end{aligned}$$

The dynamic semantics of this calculus is given by Boudol's reduction rules [5]. Although they implement a call-by-value strategy, these rules are able to evaluate correctly recursive definitions involving function applications, such as:

$$\begin{aligned}\textbf{let rec } x = (\lambda yz.(zy))x \textbf{ in } x &\to \textbf{let rec } x = \lambda z.(zx) \textbf{ in } x\\ &\to \textbf{let rec } x = \lambda z.(zx) \textbf{ in } \lambda z.(zx)\\ &\to \lambda z.(z(\textbf{let rec } x = \lambda z.(zx) \textbf{ in } \lambda z.(zx)))\end{aligned}$$

The dynamic semantics of the calculus is defined in figure 6. Technically, the semantics is a very simple call-by-value evaluation, with the only non-standard point that variables are considered values, allowing to reduce terms like the one above. Note that the parallel capture-avoiding substitution must be defined in order to properly write the (mutrec) rule. Its definition is standard.

4.3 The translation

The translation scheme for our language is defined in figure 7. The translation is type-directed and operates on terms annotated by their types. For the core language constructs (variables, constants, abstractions, applications), the translation is a simple morphism; the corresponding cases are omitted from figure 7.

Access to a structure component $E.X$ is translated into an access to field X of the record obtained by translating E. Conversely, a structure $\langle \iota; o \rangle$ is translated into a record construction. The resulting record has one field for each exported name $X \in dom(o)$, and this field is associated to $o(X)$ where all input parameters on which X depends are λ-abstracted. Some notation is required here. We write $\mathcal{D}^{-1}(X)$ for the list of immediate predecessors of node X in the dependency graph $\mathcal{D}$, ordered lexicographically. (The ordering is needed to ensure that values for these predecessors are provided in the correct order later; any fixed total ordering will do.) If $(X_1, \ldots, X_n) = \mathcal{D}^{-1}(X)$ is such a list, we write $\iota^{-1}(\mathcal{D}^{-1}(X))$ for the list $(x_1, \ldots, x_n)$ of variables associated to the names $(X_1, \ldots, X_n)$ by the input mapping ι. Finally, we write $\boldsymbol{\lambda}(x_1, \ldots, x_n).M$ as shorthand for $\lambda x_1 \ldots \lambda x_n.M$. With all this notation, the field X in the record translating $\langle \iota; o \rangle$ is bound to $\boldsymbol{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X)).[\![o(X) : \mathcal{O}(X)]\!]$.

The sum of two mixins $E_1 + E_2$ is translated by building a record containing the union of the fields of the translations of E_1 and E_2. For the delete operator $E \setminus X$, we return a copy of the record representing E in which the field X is omitted. Renaming $E[X \leftarrow Y]$ is harder: not only do we need to rename the field X of the record representing E into Y, but the renaming of X to Y in the input parameters can cause the order of the implicit arguments of the record fields to change. Thus, we need to abstract again over these parameters in the correct order after the renaming, then apply the corresponding field of $[\![E]\!]$ to these parameters in the correct order before the renaming. Again, some notation is in order: to each name X we associate a fresh variable written $\overline{X}$, and similarly for lists of names, which become lists of variables. Moreover, we write $M\ (x_1, \ldots, x_n)$ as shorthand for $M\ x_1\ \ldots\ x_n$.

The freeze operation $E\,!\,X$ is perhaps the hardest to compile. Output components Z that do not depend on X are simply re-exported from $[\![E]\!]$. For the other output components, consider a component Y of E that depends on

$Y_1, \ldots, Y_n$, and assume that one of these dependencies is X, which itself depends on $X_1, \ldots, X_p$. In $E \,!\, X$, the Y component depends on $(\{Y_i\} \cup \{X_j\}) \setminus \{X\}$. Thus, we λ-abstract on the corresponding variables, then compute X by applying $[\![E]\!].X$ to the parameters $\overline{X_j}$. Since X can depend on itself, this application must be done in a **let rec** binding over $\overline{X}$. Then, we apply $[\![E]\!].Y$ to the parameters that it expects, namely $\overline{Y_i}$, which include $\overline{X}$.

The only operator that remains to be explained is $\mathtt{close}(E)$. Here, we take advantage of the fact that `close` removes all input dependencies to generate code that is more efficient than a sequence of freeze operations. We first *serialize* the set of names exported by E against its dependency graph $\mathcal{D}$. That is, we identify strongly connected components of $\mathcal{D}$, then sort them in topological order. The result is an enumeration $(\{X_1^1 \ldots X_{n_1}^1\}, \ldots, \{X_1^p \ldots X_{n_p}^p\})$ of the exported names where each cluster $\{X_1^i \ldots X_{n_i}^i\}$ represents mutually recursive definitions, and the clusters are listed in an order such that each cluster depends only on the preceding ones. We then generate a sequence of **let rec** bindings, one for each cluster, in the order above. In the end, all output components are bound to values with no dependencies, and can be grouped together in a record.

5 Type soundness of the translation

5.1 A type system for the target language

The translation scheme defined above can generate recursive definitions of the form **let rec** $x = M\ x$ **in** $\ldots$. In λ_B, these definitions can either evaluate to a fixpoint (*i.e.* $M = \lambda x.\lambda y.y$), or get stuck (*i.e.* $M = \lambda x.x + 1$). In preparation for showing that no term generated by the translation can get stuck, we now equip λ_B with a sound type system that guarantees that all recursive definitions are correct. Boudol [5] gave such a type system, however it does not type-check curried function applications with sufficient precision for our purposes. Hence we now define a refinement of Boudol's type system.

The type system for λ_B is defined in figure 8. Types, written τ, have the following syntax:

$\tau ::= \mathtt{int} \mid \mathtt{bool}$	base types
$\mid \tau_1 \xrightarrow{d} \tau_2$	annotated function types
$\mid \langle \ldots\ X_i : \tau_i\ \ldots \rangle$	record types

Arrow types are annotated with *degrees* d, indicating how a function uses its argument. For instance, a function such as $\lambda x.x + 1$ has type $\mathtt{int} \xrightarrow{0} \mathtt{int}$, because the value of x is immediately needed after application, whereas $\lambda xyz.x + 1$ has type $\mathtt{int} \xrightarrow{2} \mathtt{int}$, because the value of x is not needed unless at least 2 more function applications are performed. Formally, a degree can be either a natural number or ∞, meaning that the variable is not used. Similarly, the typing judgment is of the form $\Gamma \vdash M : \tau \,/\, \gamma$, where γ is a (total) mapping from variables to degrees, indicating how M uses each variable: $\gamma(x) = \infty$ means that x is not free in M; $\gamma(x) = 0$ means that the value of x is needed to evaluate M; and

$\gamma(x) = n + 1$ means that the value of x is needed only after $n + 1$ function applications, *e.g.* x occurs in M under at least $n+1$ function abstractions.

Rule (var) expresses that the variable x is immediately used via the side condition $\gamma(x) = 0$. Function abstraction (rule (abstr)) increments by 1 the degree of all variables appearing in its body, except for its formal parameter x, whose degree is retained in the type of the function. We write $\gamma - 1$ for the function $y \mapsto \gamma(y) - 1$, with the convention that $0 - 1 = 0$ and $\infty - 1 = \infty$. We write $(\gamma - 1)[x \mapsto d]$ for the function that maps x to d, and otherwise behaves like $(\gamma - 1)$.

Rule (app) deals with general function application. In the function part M_1, all variable degrees are decremented by 1, since the application removes one level of abstraction. The degrees of the argument part M_2 are combined with the d annotation on the arrow type of M_1 via the @ operation, defined as follows:

$$d \mathbin{@} 0 = 0 \qquad d \mathbin{@} \infty = \infty \qquad d \mathbin{@} (n+1) = d$$

Because of call-by-value, immediate dependencies in M_2 ($\gamma_2(x) = 0$) are still immediate in the application. Variables not free in M_2 ($\gamma_2(x) = \infty$) do not contribute any dependency to the application. The interesting case is that of a variable x with degree $n+1$ in M_2, *i.e.* not immediately needed. We do not know how many times the function M_1 is going to apply its argument inside its body. However, we know that it will not do so before d more applications of $M_1\ M_2$. Hence, we can take d for the degree of x in $M_1\ M_2$. Finally, the contributions from the function part ($\gamma_1 - 1$) and the argument part ($d \mathbin{@} \gamma_2$) are combined with the $\wedge$ operator, which is point-wise minimum.

When the argument of an application is a variable, as in $M\ x$, a more precise type-checking is possible (rule (appvar)). Namely, the variable x is not needed immediately, but only when the function M needs its argument. Hence, the degree of x in the application is $(\gamma(x) - 1) \wedge d$, while all other variables y have degree $\gamma(y) - 1$.

The most complex rule is (rec) for mutual recursive definitions. Intuitively, the right-hand sides $M_1 \ldots M_n$ must not depend immediately on any of the recursively defined variables $x_1 \ldots x_n$. In other terms, the dependency d_{ij} of M_i on x_j must satisfy $d_{ij} \geq 1$. However, we must also take into account indirect dependencies: for instance, M_1 may depend on x_2, whose definition M_2 in turn depends on x_3, making M_1 depend on x_3 as well. We account for these indirect dependencies via the triangular inequality $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$. Finally, the dependencies of the whole **let rec** are obtained by combining those of its body M with those arising from the uses of the x_i in M, either direct ($d_i \mathbin{@} \gamma_i$) or one-step indirect ($d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j$). Longer indirect dependencies such as $d_i \mathbin{@} d_{ij} \mathbin{@} d_{jk} \mathbin{@} \gamma_k$ need not be taken into account because of the triangular inequality.

Finally, the (let) rule is a combination of the (abstr) and (app) rules, and the rules for record operations (record) and (sel) are straightforward.

Theorem 1 (soundness of λ_B) *If $\Gamma \vdash M : \tau \ /\ \gamma$ and $\gamma(x) \geq 1$ for all x free in M, then M either reduces to a value or diverges, but does not get stuck.*

Proof The proof is in appendix B. Basically, the principal steps are the two following substitution lemmas, one for each application rule. First, a more or less standard lemma for the (var) rule.

Lemma 3 (substitution) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau \;/\; \gamma_1[x \mapsto d]$*, and* $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2$*, with* $x \notin FV(M_2) \cup dom(\gamma_2)$*, then* $\Gamma \vdash M_1\{x \leftarrow M_2\} : \tau \;/\; \gamma_1 \wedge d \,@\, \gamma_2$*.*

Then, a specialized lemma for the (appvar) rule.

Lemma 4 (substitution by a variable) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \;/\; \gamma[x \mapsto d]$ *and* $\Gamma(y) = \tau'$*, then* $\Gamma \vdash M\{x \leftarrow y\} : \tau \;/\; \gamma \wedge (y \mapsto d)$*.*

Then a lemma for parallel substitution is stated.

Lemma 5 (parallel substitution) *If* $\Gamma + \{\ldots\; x_i{:}\tau_i\; \ldots\} \vdash M : \tau \;/\; \gamma_M[\ldots\; x_i \mapsto d_i\; \ldots]$*, and for all* $j \in \{1 \ldots n\}$*,* $\Gamma \vdash M_j : \tau_j \;/\; \gamma_j$ *with for all* i, j*,* $x_i \notin FV(M_j) \cup dom(\gamma_j)$*, then* $\Gamma \vdash M\{\ldots\; x_i \leftarrow M_i\; \ldots\} : \tau \;/\; \gamma_M \wedge \bigwedge_i d_i \,@\, \gamma_i$*.*

Here the harder is done and we can state the traditional subject-reduction and progress properties.

Lemma 6 (subject-reduction) *If* $\Gamma \vdash M : \tau \;/\; \gamma$ *and* $M \to M'$*, then* $\Gamma \vdash M : \tau \;/\; \gamma$*.*

Lemma 7 (Progress) *If* $\Gamma \vdash M : \tau \;/\; \gamma$ *and* $\gamma \geq 1$*, then either* M *is a value, or there exists* M' *such that* $M \to M'$*.*

$\square$

5.2 Soundness of the translation

The goal of this section is to prove the soundness of our approach, in the sense that a well-typed CMS_v expression is translated as a well-typed λ_B expression, the soundness of λ_B ensuring then that its evaluation does not get stuck. Being able to state the soundness of the translation implies that we have a translation for types. Before to set it up, we define some ways to deal with graphs and signatures in figure 9. We define $FCT_{\mathcal{D}}(X, \mathcal{I})$ as the list of the types and valuations of the predecessors of X in $\mathcal{D}$ according to $\mathcal{I}$, ordered lexicographically. Then $Pred(\mathcal{D})$ and $Succ(\mathcal{D})$ are simply the sets of predecessors and successors of any node in $\mathcal{D}$. The translation of types is presented in figure 10. A natural translation for environments follows, defined by $[\![\Gamma]\!] = [\![\cdot]\!] \circ \Gamma$. Moreover, we define the initial degree environment corresponding to a type environment as $d^o(\Gamma) = \underline{0} \circ \Gamma$, that is to say the function equal to 0 on $dom(\Gamma)$ and ∞ elsewhere. In the sequel, we will often use valuations as degrees. It is worth noticing that for all valuations χ_1, and χ_2, $\min(\chi_1, \chi_2) = \chi_1 \wedge \chi_2 = \chi_1 \,@\, \chi_2$.

As the translation operates on annotated well-typed terms, we define an annotated syntax in figure 11 The type system for annotated terms is exactly the same, except that it looks more like a well-formedness judgment $\Gamma \vdash \overline{E}$. Thus a derivation for a standard term yields a correct derivation for the corresponding annotated term. We denote by $\overline{E}$ the annotated term corresponding to a derivation of E, which should be clear from the context. A well-formed annotated term is a term whose annotations are all well-formed types. We consider only well-formed expressions.

The proof is detailed in section D, so we just give an outline here. It begins with two expressivity results for λ_B. First, it deals with successive abstractions as expected.

Lemma 8 (n abstractions) *If* $\Gamma + \{\dots\ x_i : \tau_i\ \dots\} \vdash M : \tau\ /\ (\gamma - n)[\dots\ x_i \mapsto d_i\ \dots]$, *then* $\Gamma \vdash \boldsymbol{\lambda}(x_1, \dots, x_n).M : \tau_1 \xrightarrow{d_1+(n-1)} \tau_2 \xrightarrow{d_2+(n-2)} \dots \tau_n \xrightarrow{d_n} \tau\ /\ \gamma$.

Second, it deals with successive applications of variables as expected.

Lemma 9 (n applications) *If* $\Gamma \vdash M : \tau_1 \xrightarrow{d_1+(n-1)} \tau_2 \xrightarrow{d_2+(n-2)} \dots \tau_n \xrightarrow{d_n} \tau\ /\ \gamma$, *and for all* $i \in \{1 \dots n\}$, $\Gamma(x_i) = \tau_i$, *then* $\Gamma \vdash M(x_1, \dots, x_n) : \tau\ /\ (\gamma - n) \wedge (\dots\ x_i \mapsto d_i\ \dots)$.

Then, there is a technical lemma to prove that the serialization technique of the translation is consistent. We skip it here because of its complex form, but the idea is that the subterms of the `let rec` sequence generated by the `close`() are well-typed in the right type and degree environments.

Then, the following theorem is proved.

Theorem 2 (soundness of the translation) *If* $\Gamma \vdash E : \mathcal{T}$, *then* $[\![\Gamma]\!] \vdash [\![\overline{E}]\!] : [\![\mathcal{T}]\!]\ /\ d^o(\Gamma) + \mathit{IsRec}(E)$.

There are two remarks to do about its statement. First, the context is not supposed to be empty, which is promising for the compatibility of the tranlsation with separate compilation (the compositionality of the translation is a better argument). Second, if the translated term is an abstraction, the degree environment may be set to 1, as is natural.

6 Related work

Mixin-based language designs Bracha [7, 6] introduced the concept of mixin as a generalization of (multiple) inheritance in class-based OO languages, allowing more freedom in deferring the definition of a method in a class and implementing it later in another class than is normally possible with inheritance and overriding.

Duggan and Sourelis [12, 13] were the first to transpose Bracha's mixin concept to the ML module system. Their mixin module system supports extensible functions and datatypes: a function defined by cases (pattern-matching) can be split across several mixins, each mixin defining only certain cases, and similarly a datatype (sum type) can be split across several mixins, each mixin defining only certain constructors; a composition operator then stitches together these cases and constructors. The problem with ill-founded recursions is avoided by allowing only functions (λ-abstractions) in the combinable parts of mixins, while initialization code goes into a separate, non-combinable part of mixins. Their compilation scheme (into ML modules) is less efficient than ours, since the fixpoint defining a function is computed at each call, rather than only once at mixin combination time as in our system. More precisely, where we generate `let rec f = A.f g and g = A.g f` using λ_B's special `let rec`, Duggan and Sourelis must generate the less efficient `let rec f x = A.f g x and g x = A.g f x`.

The *units* of Flatt and Felleisen [14] are a module system for Scheme. The basic program units import names and export definitions, much like in Ancona and Zucca's *CMS* calculus. The recursion problem is solved as in [12] by separating initialization from component definition.

Mixin calculi Ancona and Zucca [1–3] develop a theory of mixins, abstracting over much of the core language, and show that it can encode the pure λ-calculus, as well as Abadi and Cardelli's object calculus. The emphasis is on providing a calculus, with reduction rules but no fixed reduction strategy, and nice confluence properties.

Another calculus of mixins is Vestergaard and Wells' `m`-calculus [19], which is very similar to *CMS* in many points, but is not based on any core language, using only variables instead. The emphasis is put on the equational theory, allowing for example to replace some variables with their definition inside a structure, or to garbage collect unused components, yielding a powerful theory.

Neither Ancona-Zucca nor Vestergaard-Wells attempt to control recursive definitions statically, performing on-demand unwinding instead. Still, some care is required when unwinding definitions inside a structure, because of confluence problems [4].

Recursive modules in ML Crary *et al* [10, 11] and Russo [18] extend the Standard ML module system with mutually recursive structures via a `structure rec A = ... and B = ...` binding. Like mixins, this construct addresses ML's cross-module recursion problem; unlike mixins, it does not support late binding and incremental programming. The `structure rec` binding does not lend itself directly to separate compilation (the definitions of all mutually recursive modules must reside in the same source file), although some amount of separate compilation can be achieved by functorizing each recursive module over the others.

ML structures contain type components in addition to value components, and this raises delicate static typing issues that we have not yet addressed within our CMS_v framework. Crary *et al* formalize static typing of recursive structure using recursively-defined signatures and the phase distinction calculus, while Russo remains closer to Standard ML's static semantics.

Concerning ill-founded recursive value definitions, Russo does not attempt to detect them statically, relying on lazy evaluation to catch them at run-time. Crary *et al* statically require that all components of recursive structures are syntactic values. This is safe, but less flexible than our component-per-component dependency analysis.

7 Conclusions and future work

As a first step towards a full mixin module system for ML, we have developed a call-by-value variant of Ancona and Zucca's calculus of mixins. The main technical innovation of our work is the use of dependency graphs in mixin signatures, statically guaranteeing that cross-module recursive definitions are well founded, yet leaving maximal flexibility in mixing recursive function definitions and non-recursive computations within a single mixin. Dependency graphs also allow a separate compilation scheme for mixins where fixpoints are taken as early as possible, *i.e.* during mixin initialization rather than at each component access.

A drawback of dependency graphs is that programmers must (in principle) provide them explicitly when declaring a mixin signature, *e.g.* for a deferred sub-mixin component. This could make programs quite verbose. Future work includes the design of a concrete syntax for mixin signatures that alleviate this problem in the most common cases. For instance, an unannotated, ML-like signature could be interpreted as follows: each component depends on all preceding components, with a "1" dependency if it has a function type and a "0" dependency otherwise; additional annotations would be provided for cases where this default interpretation is insufficient.

Our λ_B target calculus can be compiled efficiently down to machine code, using the "in-place updating" trick described in [9] to implement the non-standard `let rec` construct. However, this trick assumes constant-sized function closures; some work is needed to accommodate variable-sized closures as used in the OCaml compiler among others.

The next step towards mixin modules for ML is to support type definitions and declarations as components of mixins. While these type components account for most of the complexity of ML module typing, we are confident that we can extend to mixins the considerable body of type-theoretic work already done for ML modules [15, 16] and recursive modules [10, 11].

References

1. D. Ancona. *Modular formal frameworks for module systems.* PhD thesis, Universita di Pisa, 1998.
2. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *PPDP'99 - International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.
3. D. Ancona and E. Zucca. A calculus of module systems. *Journal of functional programming*, 2001. To appear.

4. Z. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 2001. To appear.
5. G. Boudol. The recursive record semantics of objects revisited. In D. Sands, editor, *ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2001.
6. G. Bracha. *The programming language Jigsaw: mixins, modularity and multiple inheritance.* PhD thesis, University of Utah, 1992.
7. G. Bracha and W. Cook. Mixin-based inheritance. In *Object-Oriented Programming Systems, Languages and Applications90*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, 1990.
8. L. Cardelli. Program fragments, linking, and modularization. In *24th symposium Principles of Programming Languages*, pages 266–277. ACM Press, 1997.
9. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.
10. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Programming Language Design and Implementation 1999*, pages 50–63. ACM Press, 1999.
11. D. Dreyer, K. Crary, and R. Harper. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, 2001.
12. D. Duggan and C. Sourelis. Mixin modules. In *International Conference on Functional Programming 96*, pages 262–273. ACM Press, 1996.
13. D. Duggan and C. Sourelis. Recursive modules and mixin-based inheritance. Unpublished draft, 2001.
14. M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Programming Language Design and Implementation 1998*, pages 236–248. ACM Press, 1998.
15. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.
16. X. Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.
17. M. Lillibridge. *Translucent sums : a foundation for higher-order module systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
18. C. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming 01*, pages 50–61, 2001.
19. J. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symp. Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 412–428. Springer-Verlag, 2000.

A Soundness of graph operations

Lemma 1 *If $\mathcal{D}$ is a safe dependency graph, then the graphs $\mathcal{D}[X \leftarrow Y]$, $\mathcal{D}\backslash X$ and $\mathcal{D}!X$ are safe.*

Proof For each operation, we show that for all path in the result graph, there exists a path with the same valuation in $\mathcal{D}$, with appropriate ends to show the soundness.

Rename Let $\mathcal{D}' = \mathcal{D}[X \leftarrow Y] = \{A\{X \leftarrow Y\} \xrightarrow{\chi} B\{X \leftarrow Y\} \mid A \xrightarrow{\chi} B \in \mathcal{D}\}$, and let P be a path of $\mathcal{D}'$, with valuation χ, and $\mathit{fst}(P) = A$ and $\mathit{last}(P) = B$.

By induction on P, we find a path with same valuation in $\mathcal{D}$, such that $\mathit{fst}(P) = A\{Y \leftarrow X\}$ and $\mathit{last}(P) = B\{Y \leftarrow X\}$.

Empty path If $P = [Z]$ for some name Z, then Z is mentioned in $\mathcal{D}'$. But all edges of $\mathcal{D}'$ are of the form $A\{X \leftarrow Y\} \xrightarrow{\chi} B\{X \leftarrow Y\}$, where the corresponding edge $A \xrightarrow{\chi} B$ is in $\mathcal{D}$. So there is a name Z' mentioned in $\mathcal{D}$ such that $Z = Z'\{X \leftarrow Y\}$. If $Z = Y$, then $Z' = X$, because Y cannot be mentioned in $\mathcal{D}$ by definition of the renaming operation, and then the path $[X]$ in $\mathcal{D}$ has same valuation as P, and the right first and last nodes. If $Z \neq Y$, then $Z = Z'$ and the path $[Z']$ of $\mathcal{D}$ has the expected valuation, first and last nodes.

Non-empty path If $P = (A \xrightarrow{\chi} B) :: P'$, then $\mathit{fst}(P') = B$. Let $\mathit{last}(P') = C$ and $\chi' = \nu(P')$. By induction hypothesis, there is a path P'' of $\mathcal{D}$, from $B\{Y \leftarrow X\}$ to $C\{Y \leftarrow X\}$, with valuation χ'. But by definition of $\mathcal{D}'$ the edge $A\{Y \leftarrow X\} \xrightarrow{\chi} B\{Y \leftarrow X\}$ is in $\mathcal{D}$, so that the path $(A\{Y \leftarrow X\} \xrightarrow{\chi} B\{Y \leftarrow X\}) :: P''$ is too. It has the expected first and last nodes, and its valuation is $\min(\chi, \chi') = \nu(P)$.

As a consequence, a cycle in $\mathcal{D}'$ corresponds to a cycle in $\mathcal{D}$, but as $\mathcal{D}$ is safe, $\mathcal{D}'$ is too.

Delete The delete operation is the easiest, since all edges of the result graph are already present in $\mathcal{D}$.

Freeze Let $\mathcal{D}' = \mathcal{D}\,!\,X = (\mathcal{D} \cup \mathcal{D}_{around}) \setminus \mathcal{D}_{remove}$, where $\mathcal{D}_{around}$ and $\mathcal{D}_{remove}$ are defined in section 3.2, and let P be a path of $\mathcal{D}'$, with valuation χ, and $\mathit{fst}(P) = A$ and $\mathit{last}(P) = B$. By induction on P, we find a path with same valuation in $\mathcal{D}$, such that $\mathit{fst}(P) = A$ and $\mathit{last}(P) = B$. In other terms, there is a path between the same names and with the same valuation as P in $\mathcal{D}$. We prove it by induction on P.

Empty path If $P = [A]$, then $A = B$. As the freezing operation does not introduce new names, all names appearing in $\mathcal{D}'$ are already in $\mathcal{D}$, so P is a path of $\mathcal{D}$ too, obviously with valuation 1.

Non-empty path If $P = (A \xrightarrow{\chi} C) :: P'$, with $\mathit{fst}(P') = C$ and by hypothesis $\mathit{last}(P') = B$. By induction hypothesis, there is a path P'' in $\mathcal{D}$ from C to B such that $\nu(P'') = \nu(P')$. Reason now by cases on the edge $A \xrightarrow{\chi} C$. By definition of the freeze operation, it can either be in $\mathcal{D}$ or in $\mathcal{D}_{around}$.

From $\mathcal{D}$: The path $A \xrightarrow{\chi} C :: P''$ is then clearly a path of $\mathcal{D}$, with the right valuation and ends.

From $\mathcal{D}_{around}$: There exist χ_1 and χ_2 such that $A \xrightarrow{\chi_1} X \in \mathcal{D}$ and $X \xrightarrow{\chi_2} C \in \mathcal{D}$ and $\chi = \min(\chi_1, \chi_2)$. So the path $(A \xrightarrow{\chi_1} X) :: (X \xrightarrow{\chi_2} C) :: P''$ is a path of $\mathcal{D}$ from A to B, with valuation $\min(\min(\chi_1, \chi_2), \nu(P'')) = \min(\chi, \nu(P')) = \nu(P)$.

□

B Soundness of the target language

We prove the soundness on a subset $\lambda_{\overline{B}}$ of λ_B, precisely we rule out the constants, the record constructs and the **let** binding, which are not difficult to add and would impede the readability of the proof. The soundness proof for $\lambda_{\overline{B}}$ follows the usual pattern, with the subject-reduction and progress properties. However, because of our two typing rules for application, we need two substitution lemmas. But before, there is a first raw of algebraic lemmas on degrees, and a few non standard lemmas related to them.

B.1 Properties of degrees

We recall first the definitions of the operations on degrees in figure 12. We then can state the following lemmas, without proving the most trivial ones. The lemmas should be read as universally quantified over the degrees d, d', d_1, d_2, d_3. We adopt the convention that @ has the highest priority, followed by $\wedge$, and then $+$ and $-$.

Lemma 10

1. $(d_1 + 1) @ d_2 \leq d_1 @ d_2 + 1$.
2. $(d_1 \wedge d_2) @ d_3 = d_1 @ d_3 \wedge d_2 @ d_3$.
3. $d_1 @ (d_2 \wedge d_3) = d_1 @ d_2 \wedge d_1 @ d_3$.
4. $(d_1 @ d_2) @ d_3 = d_1 @ (d_2 @ d_3)$.
5. $(d - n) @ d' = d @ d' - n$.
6. *If* $d + 1 = d'$,
   then $d' \geq 1$ *and* $d = d' - 1$.
7. *If* $d \neq 0$, *then* $d - 1 + 1 = d$.
8. $0 @ d \leq d$.
9. *If* $d \leq d'$ *then* $d + 1 \leq d' + 1$.
10. *If* $d + 1 \leq d' - 1$ *then* $d + 2 \leq d'$.
11. *If* $d_2 \geq 1$, *then* $d_1 @ d_3 \leq d_1 @ d_2 @ d_3$.

Proof

1. If $d_2 = 0$, we obtain $0 \leq 1$ which is true. Else, if $d_2 = \infty$ we obtain $\infty \leq \infty$, and otherwise we get $d_1 + 1 \leq d_1 + 1$.
2. If $d_3 = 0$, we obtain 0 on both sides of the equality, else if $d_3 = \infty$, both sides are equal to ∞, and otherwise we get $d_1 \wedge d_2$ on both sides.
3. If $d_2 = 0$, both sides are equal to 0. Else, if $d_2 = \infty$, then $d_2 \wedge d_3 = d_3$ and $d_1 @ d_2 = \infty$, so both members are equal to $d_1 @ d_3$. Otherwise, reason by case on d_3. If $d_3 = 0$, then we obtain 0 on both sides, and if $d_3 = \infty$, we obtain $d_1 @ d_2$ for both members. Otherwise, $d_2 \wedge d_3 = n \neq 0$, so $d_1 @ (d_2 \wedge d_3) = d_1 = d_1 \wedge d_1 = d_1 @ d_2 \wedge d_1 @ d_3$.
4. If $d_3 = 0$, both members are equal to 0, else, if $d_3 = \infty$, we obtain ∞ on both sides, and otherwise we obtain $d_1 @ d_2$.

5. If $d' = \infty$, everything makes ∞. If $d' = 0$, everything makes 0. Otherwise, we obtain $d - 1$ for both members.
6. Easy.
7. Easy.
8. Easy.
9. Easy.
10. d' cannot be 0, or else we would have $d + 1 \leq 0$ as a hypothesis. Thus, by the above properties, we get $d + 2 \leq d'$.
11. - If $d_3 = \infty$ or 0, then everything makes d_3.
    - If $d_3 = n + 1$, then $d_1 \mathbin{@} d_3 = d_1$ and $d_1 \mathbin{@} d_2 \mathbin{@} d_3 = d_1 \mathbin{@} d_2$, so we just want to prove $d_1 \leq d_1 \mathbin{@} d_2$.
      - If $d_2 = \infty$, then $d_1 \mathbin{@} d_2 = \infty$ which cannot be inferior to d_1.
      - If $d_2 = m + 1$, then $d_1 \mathbin{@} d_2 = d_1$, so it works.

□

B.2 Weakening lemmas

We now prove what corresponds to the standard weakening lemma.

Lemma 11 (degree restriction) *If $\gamma' \leq \gamma$ and $\Gamma \vdash M : \tau \;/\; \gamma$, then $\Gamma \vdash M : \tau \;/\; \gamma'$.*

The proof of this lemma uses three more simple lemmas on degree environments, which are put in appendix C.

Proof We reason by induction on the typing derivation of M.

var $M = x$. We know that $\Gamma(x) = \tau$ and $\gamma(x) = 0 \geq \gamma'(x)$, so $\gamma'(x) = 0$ and we can build the axiom again.

abstr $M = \lambda x.M_1$. By typing we have a derivation of $\Gamma + \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 \;/\; (\gamma - 1)[x \mapsto d]$ with $\tau = \tau_1 \xrightarrow{d} \tau_2$. But it is easy to notice that $(\gamma' - 1)[x \mapsto d] \leq (\gamma - 1)[x \mapsto d]$, so by induction hypothesis, we have a derivation of $\Gamma + \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 \;/\; (\gamma' - 1)[x \mapsto d]$, which allows to recover the expected judgment by another application of the rule (abstr).

app $M = M_1\ M_2$. By typing we have derivations for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2$, with $\gamma = (\gamma_1 - 1) \wedge d@\gamma_2$. By lemma 16, we find γ_1' and γ_2', such that $\gamma_1' \leq \gamma_1$, $\gamma_2' \leq \gamma_2$ and $\gamma' = (\gamma_1' - 1) \wedge d@\gamma_2'$. By induction hypothesis, we get derivations for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1'$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2'$, and we can apply the rule (app) again to obtain the desired judgment.

appvar $M = M_1 x$. By typing we have a derivation for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1$ with $\Gamma(x) = \tau'$ and $\gamma = (\gamma_1 - 1) \wedge d$. So $\gamma' \leq (\gamma_1 - 1) \wedge (x \mapsto d)$, which gives us by lemma 17 a γ_1' such that $\gamma_1' \leq \gamma_1$ and $\gamma' = (\gamma_1' - 1) \wedge (x \mapsto d)$. Similarly, we can apply rule (appx) again to derive the expected judgment.

rec $M = \textbf{let rec} \;\ldots\; x_i = M_i \;\ldots\; \textbf{in}\ N$. By typing we have
- $\Gamma + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \;/\; \gamma_0[\ldots\ x_j \mapsto d_j\ \ldots]$
- $\Gamma + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash M_i : \tau_i \;/\; \gamma_i[\ldots\ x_j \mapsto d_{ij}\ \ldots]$

  - $\forall i, j : \ d_{ij} \geq 1$
  - $\forall i, j, k : \ d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$

  and $\gamma = \gamma_0 \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$. Looking at the proof of the lemma 18, we set $\gamma'_N = \gamma'$ and for all i, $\gamma'_i = \gamma_i$, knowing that $\gamma'_N \leq \gamma_0$ and $\gamma' = \gamma'_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big)$. By induction hypothesis, we know how to derive $\Gamma + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \ / \ \gamma'_N[\ldots\ x_j \mapsto d_j\ \ \ldots]$, so we can derive $\Gamma \vdash M : \tau \ / \ \gamma'$ too.

□

Lemma 12 (degree weakening) *If $\Gamma \vdash M : \tau \ / \ \gamma[x \mapsto d]$ and $x \notin FV(M)$, then $\Gamma \vdash M : \tau \ / \ \gamma$.*

Again, this lemma requires some more lemmas, those of appendix C.2.

Proof We reason by induction on the typing derivation of M.

var $M = y$. As $x \notin FV(M)$, $x \neq y$, so by typing $\gamma(y) = 0$ and $\Gamma(y) = \tau$ and $\Gamma \vdash M : \tau \ / \ \gamma$.

abstr $M = \lambda y.M_1$, with a fresh y. By typing we have $\Gamma + \{y \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma[x \mapsto d] - 1)[y \mapsto d_0]$ and $\tau = \tau_1 \xrightarrow{d_0} \tau_2$. But obviously $(\gamma[x \mapsto d] - 1)[y \mapsto d_0] = (\gamma - 1)[y \mapsto d_0][x \mapsto d - 1]$, so by induction hypothesis we get $\Gamma + \{y \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma - 1)[y \mapsto d_0]$ and by rule (abstr) we recover the expected judgment.

app $M = M_1\ M_2$. By typing we have

$$\Gamma \vdash M_1 : \tau' \xrightarrow{d_0} \tau \ / \ \gamma_1$$

$$\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$$

with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$. So by lemma 19, we find d_1, d_2, γ'_1 and γ'_2 such that $\gamma = (\gamma'_1 - 1) \wedge d_0 \mathbin{@} \gamma'_2$, $\gamma'_1[x \mapsto d_1] = \gamma_1$ and $\gamma'_2[x \mapsto d_2] = \gamma_2$. By induction hypothesis we can derive

$$\Gamma \vdash M_1 : \tau' \xrightarrow{d_0} \tau \ / \ \gamma'_1$$

and

$$\Gamma \vdash M_2 : \tau' \ / \ \gamma'_2$$

and by rule (app) the expected judgment.

appvar $M = M_1 y$, with $y \neq x$ by hypothesis. By typing we can derive $\Gamma \vdash M_1 : \tau_1 \xrightarrow{d_0} \tau_2 \ / \ \gamma_1$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge (y \mapsto d_0)$. Let $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. we have $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$ and $\gamma = (\gamma'_1 - 1) \wedge (y \mapsto d_0)$. Indeed, the first statement is easy to prove and for the second one, on x, it amounts to $\gamma(x) = \gamma(x) + 1 - 1$, and on another variable, as $((\gamma_1 - 1) \wedge (y \mapsto d_0))(z) = ((\gamma'_1 - 1) \wedge (y \mapsto d_0))(z)$ we get the expected result.

Then we conclude by induction hypothesis as above.

rec $M = \textbf{let rec} \ \ldots \ x_i = M_i \ \ldots \ \textbf{in} \ N$. By typing we have

$$\Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash N : \tau \ / \ \gamma_N[\ldots \ x_j \mapsto d_j \ \ldots]$$

and for all i

$$\Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash M_i : \tau_i \ / \ \gamma_i[\ldots \ x_j \mapsto d_{ij} \ \ldots]$$

with for all i, j, k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$ and for all i, j, $d_{ij} \geq 1$ and $\gamma[x \mapsto d] = \gamma_N \wedge \left(\bigwedge_i d_i \mathbin{@} \gamma_i\right) \wedge \left(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\right)$. The lemma 20 gives us γ'_N and a γ'_i for all i, such that $\gamma'_N[x \mapsto d_N] = \gamma_N$ and for all i $\gamma'_i[x \mapsto d'_i] = \gamma_i$ and $\gamma = \gamma'_N \wedge \left(\bigwedge_i d_i \mathbin{@} \gamma'_i\right) \wedge \left(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\right)$, with $d_N = \gamma_N(x)$ and for all i, $d'_i = \gamma'_i(x)$. By induction hypothesis, we derive

$$\Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash N : \tau \ / \ \gamma'_N[\ldots \ x_j \mapsto d_j \ \ldots]$$

and for all i

$$\Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash M_i : \tau_i \ / \ \gamma'_i[\ldots \ x_j \mapsto d_{ij} \ \ldots]$$

So we get the result by rule (rec).

□

Lemma 13 (type weakening) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ / \ \gamma$ *and* $x \notin FV(M)$, *then* $\Gamma \vdash M : \tau \ / \ \gamma$.

Proof Easy proof by induction on the typing derivation. □

B.3 Substitution

The most difficult lemma is the traditional substitution lemma, which is a first link between our definition of @ and what really happens during the reduction.

Lemma 3 (substitution) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau \ / \ \gamma_1[x \mapsto d]$, *and* $\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$, *with* $x \notin FV(M_2) \cup dom(\gamma_2)$, *then* $\Gamma \vdash M_1\{x \leftarrow M_2\} : \tau \ / \ \gamma_1 \wedge d \mathbin{@} \gamma_2$.

Proof We proceed by induction on the typing derivation of M_1, letting $M = M_1\{x \leftarrow M_2\}$, $\Gamma' = \Gamma + \{x \mapsto \tau'\}$, and $\gamma_0 = \gamma_1 \wedge d \mathbin{@} \gamma_2$.

var $M_1 = y$, and $\Gamma'(y) = \tau$ and $\gamma_1[x \mapsto d](y) = 0$.

- If $y = x$, then $M = M_2$, $d = 0$, $\tau = \tau'$ and by hypothesis $\Gamma \vdash M : \tau \ / \ \gamma_2$. So by lemma 11, it is enough that $\gamma_0 \leq \gamma_2$, or $\gamma_1 \wedge 0 \mathbin{@} \gamma_2 \leq \gamma_2$, which is true by lemma 10.
- If $y \neq x$, then $x \notin FV(M)$ and $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ / \ \gamma_1[x \mapsto d]$, so by lemmas 12 and 13, $\Gamma \vdash M : \tau \ / \ \gamma_1$, and it is enough that $\gamma_0 \leq \gamma_1$, which is trivially true.

abstr $M_1 = \lambda y.M_3$ (with y fresh). By typing, we know

$$\Gamma' + \{y \mapsto \tau_1\} \vdash M_3 : \tau_2 \;/\; \gamma_3[y \mapsto d_0]$$

with $\tau = \tau_1 \xrightarrow{d_0} \tau_2$ and $\gamma_3[y \mapsto d_0] = (\gamma_1[x \mapsto d] - 1)[y \mapsto d_0] = (\gamma_1 - 1)[x \mapsto (d-1); y \mapsto d_0]$. Let $M_3' = M_3\{x \leftarrow M_2\}$, by induction hypothesis, we have $\Gamma + \{y \mapsto \tau_1\} \vdash M_3' : \tau_2 \;/\; (\gamma_1 - 1)[y \mapsto d_0] \wedge (d-1) \mathbin{@} \gamma_2$. As y is fresh, it doesn't appear in γ_2, so

$$\begin{aligned}
&(\gamma_1 - 1)[y \mapsto d_0] \wedge (d-1) \mathbin{@} \gamma_2 \\
&= ((\gamma_1 - 1) \wedge (d-1) \mathbin{@} \gamma_2)[y \mapsto d_0] \\
&= ((\gamma_1 - 1) \wedge (d \mathbin{@} \gamma_2 - 1))[y \mapsto d_0] \text{ by lemma 10} \\
&= ((\gamma_1 \wedge d \mathbin{@} \gamma_2) - 1)[y \mapsto d_0] = (\gamma_0 - 1)[y \mapsto d_0]
\end{aligned}$$

so we can derive by rule (abstr) $\Gamma \vdash \lambda y.M_3' : \tau_1 \xrightarrow{d_0} \tau_2 \;/\; \gamma_0$ which is what we were expecting.

app $M_1 = M_3\ M_4$. By typing we have $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_3$ and $\Gamma' \vdash M_4 : \tau'' \;/\; \gamma_4$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge d_0 \mathbin{@} \gamma_4$. By lemma 19, if $d_3 = \gamma_3(x)$ and $d_4 = \gamma_4(x)$, we find γ_3' and γ_4' such that $\gamma_3'[x \mapsto d_3] = \gamma_3$, $\gamma_4'[x \mapsto d_4] = \gamma_4$, and $\gamma_1 = (\gamma_3' - 1) \wedge d_0 \mathbin{@} \gamma_4'$. By induction hypothesis, if $M_3' = M_3\{x \leftarrow M_2\}$ and $M_4' = M_4\{x \leftarrow M_2\}$, then $\Gamma \vdash M_3' : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_3' \wedge d_3 \mathbin{@} \gamma_2$ and $\Gamma \vdash M_4' : \tau'' \;/\; \gamma_4' \wedge d_4 \mathbin{@} \gamma_2$, so by rule (app)

$$\Gamma \vdash M : \tau \;/\; ((\gamma_3' \wedge d_3 \mathbin{@} \gamma_2) - 1) \wedge d_0 \mathbin{@} (\gamma_4' \wedge d_4 \mathbin{@} \gamma_2)$$

But by lemma 10, the degree environment is equal to

$$\begin{aligned}
&(\gamma_3' - 1) \wedge (d_3 \mathbin{@} \gamma_2 - 1) \wedge (d_0 \mathbin{@} \gamma_4') \wedge (d_0 \mathbin{@} d_4 \mathbin{@} \gamma_2) \\
&= \gamma_1 \wedge (d_3 \mathbin{@} \gamma_2 - 1) \wedge (d_0 \mathbin{@} d_4 \mathbin{@} \gamma_2) \\
&= \gamma_1 \wedge ((d_3 - 1_{\wedge} d_0 \mathbin{@} d_4) \mathbin{@} \gamma_2 \\
&= \gamma_1 \wedge d \mathbin{@} \gamma_2 \\
&= \gamma_0
\end{aligned}$$

appvar $M_1 = M_3 y$. As above, we split the analysis according to whether $y = x$ or not.

- $y = x$. Then $M = M_3'\ M_2$, where $M_3' = M_3\{x \leftarrow M_2\}$. Typing gives $(*)\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_3$, $\Gamma'(y) = \Gamma'(x) = \tau' = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Let now $\gamma_3' = \gamma_3[x \mapsto \gamma_1(x) + 1]$, we have $\gamma_1 = (\gamma_3' - 1)$ easily, and $\gamma_3'[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we can write the judgment (*) like this

  $$\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_3'[x \mapsto \gamma_3(x)]$$

  so that by induction hypothesis, we have

  $$\Gamma \vdash M_3' : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_3' \wedge d_3 \mathbin{@} \gamma_2$$

  with $d_3 = \gamma_3(x)$. Then by rule (app), we get

  $$\Gamma \vdash M : \tau \;/\; ((\gamma_3' \wedge d_3 \mathbin{@} \gamma_2) - 1) \wedge d_0 \mathbin{@} \gamma_2$$

But

$$\gamma_0 = (\gamma_3' - 1) \wedge d \mathbin{@} \gamma_2$$

and as $d = (d_3 - 1) \wedge d_0$, it reduces to

$$\gamma_0 = (\gamma_3' - 1) \wedge (d_3 \mathbin{@} \gamma_2 - 1) \wedge d_0 \mathbin{@} \gamma_2$$

so we have derived the desired judgment.

- $y \neq x$. Then $M = M_3'\ y$, where $M_3' = M_3\{x \leftarrow M_2\}$. Typing gives (*)$\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ /\ \gamma_3$, $\Gamma'(y) = \Gamma(y) = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Let now $\gamma_3' = \gamma_3[x \mapsto \gamma_1(x) + 1]$, we have $\gamma_1 = (\gamma_3' - 1) \wedge (y \mapsto d_0)$ easily, and $\gamma_3'[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we can write the judgment (*) like this

  $$\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ /\ \gamma_3'[x \mapsto \gamma_3(x)]$$

  so that by induction hypothesis, we have

  $$\Gamma \vdash M_3' : \tau'' \xrightarrow{d_0} \tau \ /\ \gamma_3' \wedge d_3 \mathbin{@} \gamma_2$$

  with $d_3 = \gamma_3(x)$. Then by rule (appvar), we get

  $$\Gamma \vdash M : \tau \ /\ ((\gamma_3' \wedge d_3 \mathbin{@} \gamma_2) - 1) \wedge (y \mapsto d_0)$$

  which yields by lemma 10

  $$\Gamma \vdash M : \tau \ /\ (\gamma_3' - 1) \wedge (d_3 \mathbin{@} \gamma_2 - 1) \wedge (y \mapsto d_0)$$

  But

  $$\begin{aligned} \gamma_0 &= \gamma_1 \wedge d \mathbin{@} \gamma_2 \\ &= (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge d \mathbin{@} \gamma_2 \\ &= (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge (d_3 - 1) \mathbin{@} \gamma_2 \\ &\qquad \text{(because } \gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)\text{)} \\ &= (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge (d_3 \mathbin{@} \gamma_2 - 1) \text{(by lemma 10)} \end{aligned}$$

  So it works too.

rec $M = \textbf{let rec}\ x_1 = N_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = N_n\ \textbf{in}\ N$ with fresh x_is and by typing
  - $\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \ /\ \gamma_N[\ldots\ x_j \mapsto d_j\ \ldots]$
  - for all i, $\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N_i : \tau_i \ /\ \delta_i[\ldots\ x_j \mapsto d_{ij}\ \ldots]$
  - for all i, j, $d_{ij} \geq 1$
  - for all i, j, k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$

  Let $N' = N\{x \leftarrow M_2\}$ and for all i, $N_i' = N_i\{x \leftarrow M_2\}$. We have $\gamma_1[x \mapsto d] = \gamma_N \wedge \big(\bigwedge\limits_i d_i \mathbin{@} \delta_i\big) \wedge \big(\bigwedge\limits_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j\big)$. Lemma 20 gives γ_N' and a δ_i' for all i, such that $\gamma_N'[x \mapsto d_N] = \gamma_N$, and $\delta_i'[x \mapsto d_i^0] = \delta_i$ for all i and $\gamma_1 = \gamma_N' \wedge \big(\bigwedge\limits_i d_i \mathbin{@} \delta_i'\big) \wedge \big(\bigwedge\limits_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j'\big)$, with $d_N = \gamma_N(x)$ and $d_i^0 = \delta_i(x)$ for each i. With this, the two premisses can be written
  - $\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \ /\ \gamma_N'[\ldots\ x_j \mapsto d_j\ \ldots][x \mapsto d_N]$

- for all i, $\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N_i : \tau_i \;/\; \delta'_i[\ldots\ x_j \mapsto d_{ij}\ \ldots][x \mapsto d^0_i]$

So by induction hypothesis

- $\Gamma + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N' : \tau \;/\; \gamma'_N[\ldots\ x_j \mapsto d_j\ \ldots] \wedge d_N \mathbin{@} \gamma_2$
- for all i, $\Gamma + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N'_i : \tau_i \;/\; \delta'_i[\ldots\ x_j \mapsto d_{ij}\ \ldots] \wedge d^0_i \mathbin{@} \gamma_2$

But as the x_is are fresh we have $\gamma'_N[\ldots\ x_j \mapsto d_j\ \ldots] \wedge d_N \mathbin{@} \gamma_2 = (\gamma'_N \wedge d_N \mathbin{@} \gamma_2)[\ldots\ x_j \mapsto d_j\ \ldots]$ and for all i, $\delta'_i[\ldots\ x_j \mapsto d_{ij}\ \ldots] \wedge d^0_i \mathbin{@} \gamma_2 = (\delta'_i \wedge d^0_i \mathbin{@} \gamma_2)[\ldots\ x_j \mapsto d_{ij}\ \ldots]$. So we can apply rule (rec) to obtain

$$\Gamma \vdash M : \tau \;/\; \gamma'_N \wedge d_N \mathbin{@} \gamma_2 \wedge \bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} (\delta'_j \wedge d^0_j \mathbin{@} \gamma_2) \wedge \bigwedge_{i} d_i \mathbin{@} (\delta'_i \wedge d^0_i \mathbin{@} \gamma_2)$$

According to the lemma 10, this big degree environment is equal to

$$\begin{aligned}
&\gamma'_N \wedge (d_N \mathbin{@} \gamma_2)\\
&\wedge(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta'_j) \wedge (\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} d^0_j \mathbin{@} \gamma_2)\\
&\wedge(\bigwedge_{i} d_i \mathbin{@} \delta'_i) \wedge (\bigwedge_{i} d_i \mathbin{@} d^0_i \mathbin{@} \gamma_2)
\end{aligned}$$

We would like it to be equal to γ_0. But as

$$\gamma_1[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_{i} d_i \mathbin{@} \delta_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j\big)$$

we know that

$$d = \gamma_N(x) \wedge \big(\bigwedge_{i} d_i \mathbin{@} \delta_i(x)\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j(x)\big)$$

and so $d = d_N \wedge \big(\bigwedge_{i} d_i \mathbin{@} d^0_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} d^0_j\big)$.

So

$$\begin{aligned}
\gamma_0 &= \gamma_1 \wedge d \mathbin{@} \gamma_2\\
&= \gamma'_N \wedge \big(\bigwedge_{i} d_i \mathbin{@} \delta'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta'_j\big)\\
&\quad \wedge\big(d_N \wedge \big(\bigwedge_{i} d_i \mathbin{@} d^0_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} d^0_j\big)\big) \mathbin{@} \gamma_2\\
&= \gamma'_N \wedge \big(\bigwedge_{i} d_i \mathbin{@} \delta'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta'_j\big)\\
&\quad \wedge(d_N \mathbin{@} \gamma_2) \wedge \big(\bigwedge_{i} d_i \mathbin{@} d^0_i \mathbin{@} \gamma_2\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} d^0_j \mathbin{@} \gamma_2\big)
\end{aligned}$$

and we are done.

□

Relying on the fact that $M\{\ldots\ x_i \leftarrow M_i\ \ldots\}$ can be written $M\{x_1 \leftarrow y_1\}\ \ldots\ \{x_n \leftarrow y_n\}\{y_1 \leftarrow M_1\}\ \ldots\ \{y_n \leftarrow M_n\}$, where the y_is are fresh, we elaborate a lemma for the parallel substitution. First we state a special single substitution lemma for variables.

Lemma 14 *If $\Gamma + \{x : \tau\} \vdash M : \tau \ / \ \gamma[x \mapsto d]$ and $y \notin FV(M)$, then $\Gamma + \{y : \tau\} \vdash M\{x \leftarrow y\} : \tau \ / \ \gamma[y \mapsto d]$.*

Proof Easy induction on the typing derivation of M. □

Then we can prove the parallel substitution lemma.

Lemma 5 *If $\Gamma + \{\ldots \ x_i : \tau_i \ \ldots\} \vdash M : \tau \ / \ \gamma_M[\ldots \ x_i \mapsto d_i \ \ldots]$, and for all $j \in \{1 \ldots n\}$, $\Gamma \vdash M_j : \tau_j \ / \ \gamma_j$ with for all i, j, $x_i \notin FV(M_j) \cup dom(\gamma_j)$, then $\Gamma \vdash M\{\ldots \ x_i \leftarrow M_i \ \ldots\} : \tau \ / \ \gamma_M \wedge \bigwedge_i d_i \mathbin{@} \gamma_i$.*

Proof Writing $M\{\ldots \ x_i \leftarrow M_i \ \ldots\}$ as

$$M\{x_1 \leftarrow y_1\} \ \ldots \ \{x_n \leftarrow y_n\}\{y_1 \leftarrow M_1\} \ \ldots \ \{y_n \leftarrow M_n\}$$

where the y_is are fresh, allows us to first to apply the lemma 14 n times to obtain $\Gamma + \{\ldots \ y_i : \tau_i \ \ldots\} \vdash M\{x_1 \leftarrow y_1\} \ \ldots \ \{x_n \leftarrow y_n\} : \tau \ / \ \gamma_M[\ldots \ y_i \mapsto d_i \ \ldots]$ and then to apply the lemma 3 n times again, successively using the n typing hypotheses for the M_is, after what we obtain the desired judgment. □

B.4 Special substitution

We prove a stronger lemma for the case where we substitute with a variable. The following lemma may appear strangely weaker than lemma 14, but here y is not supposed to be fresh, and this is why former occurences of y must be taken into account, which is done through the $\wedge$ operation.

Lemma 4 (substitution by a variable) *If $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ / \ \gamma[x \mapsto d]$ and $\Gamma(y) = \tau'$, then $\Gamma \vdash M\{x \leftarrow y\} : \tau \ / \ \gamma \wedge (y \mapsto d)$.*

Proof We proceed by induction on the typing derivation of M, letting $\Gamma' = \Gamma + \{x \mapsto \tau'\}$ and $M' = M\{x \leftarrow y\}$.

var We must consider the three cases where $M = x$, $M = y$ and $M = z$ with $z \neq x$ and $z \neq y$, but all three are easy.

abstr $M = \lambda z.M_1$, with z fresh. By typing we have

$$\Gamma' + \{z \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma[x \mapsto d] - 1)[z \mapsto d_0]$$

with $\tau = \tau_1 \xrightarrow{d_0} \tau_2$. This is equivalent to

$$\Gamma' + \{z \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma - 1)[z \mapsto d_0][x \mapsto d - 1]$$

By induction hypothesis, we then have

$$\Gamma + \{z \mapsto \tau_1\} \vdash M_1\{x \leftarrow y\} : \tau_2 \ / \ (\gamma - 1)[z \mapsto d_0] \wedge (y \mapsto d - 1)$$

which yields

$$\Gamma + \{z \mapsto \tau_1\} \vdash M_1\{x \leftarrow y\} : \tau_2 \ / \ ((\gamma \wedge (y \mapsto d)) - 1)[z \mapsto d_0]$$

and by rule (abstr)

$$\Gamma \vdash M\{x \leftarrow y\} : \tau \;/\; \gamma \wedge (y \mapsto d)$$

app $M = M_1\ M_2$. By typing we know that

- $\Gamma' \vdash M_1 : \tau' \xrightarrow{d_0} \tau \;/\; \gamma_1$
- $\Gamma' \vdash M_2 : \tau' \;/\; \gamma_2$

with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 @ \gamma_2$. Let $\gamma_1' = \gamma_1[x \mapsto \gamma(x)+1]$ and $\gamma_2' = \gamma_2[x \mapsto \infty]$. They have the three following properties:

- $\gamma_1 = \gamma_1'[x \mapsto \gamma_1(x)]$
- $\gamma_2 = \gamma_2'[x \mapsto \gamma_2(x)]$
- $\gamma = (\gamma_1' - 1) \wedge d_0 @ \gamma_2'$

So, by induction hypothesis, we can derive

$$\frac{\begin{array}{c}\Gamma \vdash M_1\{x \leftarrow y\} : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_1' \wedge (y \mapsto \gamma_1(x)) \\ \Gamma \vdash M_2\{x \leftarrow y\} : \tau'' \;/\; \gamma_2' \wedge (y \mapsto \gamma_2(x))\end{array}}{\Gamma \vdash M' : \tau \;/\; (\gamma_1' - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge d_0 @ (\gamma_2' \wedge (y \mapsto \gamma_2(x)))}$$

But the result degree environment can be seen as

$$\begin{array}{l}(\gamma_1' - 1) \wedge d_0 @ \gamma_2' \wedge (y \mapsto ((\gamma_1(x) - 1) \wedge d_0 @ \gamma_2(x))) \\ = \gamma \wedge (y \mapsto d)\end{array}$$

appvar $M = M_1\ z$ By typing we have

- $\Gamma' \vdash M_1 : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_1$
- $\Gamma'(z) = \tau''$
- $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge (z \mapsto d_0)$

We must separate the cases where $z = x$ and $z \neq x$.

- $z = x$, then $\tau' = \tau''$. Let $\gamma_1' = \gamma_1[x \mapsto \gamma(x) + 1]$, we have $\gamma_1' - 1 = \gamma$ and $\gamma_1'[x \mapsto \gamma_1(x)] = \gamma_1$. By induction hypothesis, we get $\Gamma \vdash M_1\{x \leftarrow y\} : \tau' \xrightarrow{d_0} \tau \;/\; \gamma_1' \wedge (y \mapsto \gamma_1(x))$, and as $\Gamma(y) = \tau'$, we have by rule (appvar) $\Gamma \vdash M' : \tau \;/\; (\gamma_1' - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge (y \mapsto d_0)$. But the result degree environment is equal to $(\gamma_1' - 1) \wedge (y \mapsto ((\gamma_1(x) - 1) \wedge d_0)) = \gamma \wedge (y \mapsto d)$.
- $z \neq x$. Let $\gamma_1' = \gamma_1[x \mapsto \gamma(x) + 1]$. We have $\gamma = (\gamma_1' - 1) \wedge (z \mapsto d_0)$ and $\gamma_1'[x \mapsto \gamma_1(x)] = \gamma_1$. By induction hypothesis, we have $\Gamma \vdash M_1\{x \leftarrow y\} : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_1' \wedge (y \mapsto \gamma_1(x))$. And as $\Gamma(z) = \tau''$, we derive by rule (appvar) $\Gamma \vdash M' : \tau \;/\; (\gamma_1' - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge (z \mapsto d_0)$ and this degree environment is equal to $\gamma \wedge (y \mapsto (\gamma_1(x) - 1)) = \gamma \wedge (y \mapsto d)$.

rec $M = \textbf{let rec} \ \dots\ x_i = M_i\ \dots\ \textbf{in}\ N$, with fresh x_is. By typing we have

- $\Gamma' + \{\dots\ x_i : \tau_i\ \dots\} \vdash M_j : \tau_j \;/\; \gamma_j[\dots\ x_j \mapsto d_{ji}\ \dots]$ for all j
- $\Gamma' + \{\dots\ x_i : \tau_i\ \dots\} \vdash N : \tau \;/\; \gamma_N[\dots\ x_i \mapsto d_i\ \dots]$
- For all i, j, $d_{ij} \geq 1$
- For all i, j, k, $d_{ik} \leq d_{ij} @ d_{jk}$

and $\gamma[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$. By lemma 20, we find γ'_N and for each i a γ'_i such that $\gamma = \gamma'_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big)$, $\gamma_N = \gamma'_N[x \mapsto d_N]$, and for all i, $\gamma_i = \gamma'_i[x \mapsto d'_i]$, with $d_N = \gamma_N(x)$ and for all i, $d'_i = \gamma_i(x)$. By induction hypothesis, we have the two following judgments.

- $\Gamma + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash M_j\{x \leftarrow y\} : \tau_j \ /\ \gamma'_j[\ldots\ x_i \mapsto d_{ji}\ \ldots] \wedge (y \mapsto d'_j)$ for all j
- $\Gamma + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash N\{x \leftarrow y\} : \tau \ /\ \gamma'_N[\ldots\ x_i \mapsto d_i\ \ldots] \wedge (y \mapsto d_N)$

from which we derive with rule (rec)

$$\Gamma \vdash M' : \tau \ /\ \gamma'$$

where

$$\begin{aligned}\gamma' &= \gamma'_N \wedge (y \mapsto d_N)\\ &\quad \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} (\gamma'_j \wedge (y \mapsto d'_j))\big)\\ &\quad \wedge \big(\bigwedge_i d_i \mathbin{@} (\gamma'_i \wedge (y \mapsto d'_i))\big)\\ &= \gamma \wedge \Big(y \mapsto \Big(d_N \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} d'_j\big) \wedge \big(\bigwedge_i d_i \mathbin{@} d'_i\big)\Big)\Big)\\ &= \gamma \wedge (y \mapsto d)\end{aligned}$$

□

B.5 Soundness

Before to prove the traditional subject-reduction lemma, we notice this easy lemma.

Lemma 15 *If $\Gamma + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash M_j : \tau_j \ /\ \gamma_j[\ldots\ x_i \mapsto d_{ji}\ \ldots]$ for all $j \in \{1 \ldots n\}$, and for all i,j, $d_{ij} \geq 1$ and for all i,j,k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$, then for all $i_0 \in \{1 \ldots n\}$, $\Gamma \vdash \textbf{let rec}\ \ldots\ x_i = M_i\ \ldots\ \textbf{in}\ M_{i_0} : \tau_{i_0} \ /\ \gamma_{i_0} \wedge \bigwedge_{i,j} d_{i_0 i} \mathbin{@} d_{ij} \mathbin{@} \gamma_j \wedge \bigwedge_i d_{i_0 i} \mathbin{@} \gamma_i$.*

As a corollary, using the triangular inequality, we have

Corollary 1 *In the same context,*

$$\Gamma \vdash \textbf{let rec}\ \ldots\ x_i = M_i\ \ldots\ \textbf{in}\ M_{i_0} : \tau_{i_0} \ /\ \gamma_{i_0} \wedge \bigwedge_i d_{i_0 i} \mathbin{@} \gamma_i$$

Lemma 6 (subject-reduction) *If $\Gamma \vdash M : \tau \ /\ \gamma$ and $M \to M'$, then $\Gamma \vdash M : \tau \ /\ \gamma$.*

Proof We proceed as usual by induction on the derivation of $M \to M'$.

beta $M = \lambda x.M_1\ v$. The typing derivation for M can end either with an application of the (app) rule or with the (appvar) rule.

appvar $v = y$, with x fresh. The typing derivation of M ends like this

$$\dfrac{\dfrac{\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau \;/\; (\gamma_0 - 1)[x \mapsto d]}{\Gamma \vdash \lambda x.M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_0} \qquad \Gamma(y) = \tau'}{\Gamma \vdash M : \tau \;/\; (\gamma_0 - 1) \wedge (y \mapsto d)}$$

and $\gamma = (\gamma_0 - 1) \wedge (y \mapsto d)$ and $M' = M_1\{x \leftarrow y\}$. By lemma 4, we have

$$\Gamma \vdash M' : \tau \;/\; (\gamma_0 - 1) \wedge (y \mapsto d)$$

which is precisely what we were expecting.

app The typing derivation of M ends like this

$$\dfrac{\dfrac{\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau \;/\; (\gamma_1 - 1)[x \mapsto d]}{\Gamma \vdash \lambda x.M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1} \qquad \dfrac{\vdots}{\Gamma \vdash v : \tau' \;/\; \gamma_2}}{\Gamma \vdash M : \tau \;/\; (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2}$$

$M' = M_1\{x \leftarrow v\}$, so by lemma 3, we have

$$\Gamma \vdash M' : \tau \;/\; (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$$

and this degree environments is equal to γ.

rec $M = \textbf{let rec} \ \dots\ x_i = v_i\ \dots\ \textbf{in}\ N$ with fresh x_is and $M' = M\{\dots\ x_i \leftarrow M_i\ \dots\}$, where for all i, $M_i = \textbf{let rec}\ \dots\ x_j = v_j\ \dots\ \textbf{in}\ v_i$.

By typing, we have

- $\Gamma + \{\dots\ x_j : \tau_j\ \dots\} \vdash N : \tau \;/\; \gamma_N[\dots\ x_j \mapsto d_j\ \dots]$
- for all i, $\Gamma + \{\dots\ x_j : \tau_j\ \dots\} \vdash v_i : \tau_i \;/\; \gamma_i[\dots\ x_j \mapsto d_{ij}\ \dots]$
- for all i, j, $d_{ij} \geq 1$
- for all i, j, k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$

By corollary 1, we derive for all i

$$\Gamma \vdash M_i : \tau_i \;/\; \gamma_i \wedge \bigwedge_j d_{ij} \mathbin{@} \gamma_j$$

and by lemma 5, we obtain

$$\Gamma \vdash M' : \tau \;/\; \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} (\gamma_i \wedge \bigwedge_j d_{ij} \mathbin{@} \gamma_j)\big)$$

which is exactly

$$\Gamma \vdash M' : \tau \;/\; \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{ij} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$$

context $M = \mathbb{E}[M_1]$, $M_1 \to M_1'$ and $M' = \mathbb{E}[M_1']$. There is a small case analysis on the context $\mathbb{E}$. There is no real difficulty, the only point being that in case $\mathbb{E}$ is of the form $v\ []$ and the typing derivation ends with rule (appvar), then M_1 can only be a variable, and therefore cannot reduce. In the other cases, it is easy to reconstruct the derivation with the reduced term.

□

Lemma 7 (Progress) *If $\Gamma \vdash M : \tau \ / \ \gamma$ and $\gamma \geq 1$, then either M is a value, or there exists M' such that $M \to M'$.*

Proof We again proceed by induction on the typing of M.

var and abstr If M is a value, no problem.
app $M = M_1\ M_2$. By typing we have
- $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \ / \ \gamma_1$
- $\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$

with $\gamma = (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$.
By induction hypothesis, either both M_1 and M_2 are values, or at least one of them reduces.
- If both are values, M_1 cannot be a variable, because $\gamma \geq 1$, so $\gamma_1 \geq 2$, so M_1 is an abstraction and we can apply the (beta) rule to reduce M.
- In the other cases, there is an evaluation context allowing to reduce M under it.

appvar Very similar.
rec Still easier.

□
The soundness theorem then easily follows.

C Restriction and weakening lemmas

C.1 Restriction lemmas

Lemma 16 *If $\gamma \leq (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$, then there exists γ_1' and γ_2' such that $\gamma = (\gamma_1' - 1) \wedge d \mathbin{@} \gamma_2'$ and $\gamma_1' \leq \gamma_1$ and $\gamma_2' \leq \gamma_2$.*

Proof We define γ_1' and γ_2' point-wise. Fix a variable x. Let $d' = \gamma(x)$, $d_1 = \gamma_1(x)$, $d_2 = \gamma_2(x)$. We just want d_1' and d_2' such that $d' = (d_1' - 1) \wedge d \mathbin{@} d_2'$ and $d_1' \leq d_1$ and $d_2' \leq d_2$.

- If $d' = 0$, then $d_1' = d_2' = 0$ works.
- If $d' = \infty$, then $d_1' = d_1$ and $d_2' = d_2$ works, because only ∞ is greater thatn d'.

- If $d' = n+1$, let $d'_1 = n+2$ and $d'_2 = d_2$. We hypothesis we know that $d' \leq d \mathbin{@} d_2$, so as $d'_1 - 1 = n+1 = d'$, we know that $(d'_1 - 1) \wedge d \mathbin{@} d'_2 = d'_1 - 1 = d'$. Moreover, as $d' \leq d_1 - 1$, we have that $n+1 \leq d_1 - 1$, and therefore $(d'_1 =)n+2 \leq d_1$ by lemma 10. Eventually, $d'_2 \leq d_2$ trivially.

□

Lemma 17 *If $\gamma \leq (\gamma_1 - 1) \wedge (x \mapsto d)$, then there exists γ'_1 such that $\gamma'_1 \leq \gamma_1$ and $\gamma = (\gamma'_1 - 1) \wedge (x \mapsto d)$.*

Proof We proceed identically as above. Fix a variable y and let $d' = \gamma(y)$ and $d_1 = \gamma_1(y)$. We are trying to find d'_1 such that $d'_1 \leq d_1$ and $d' = (d'_1 - 1) \wedge ((x \mapsto d)(y))$.

- If $d_1 = 0$, then $d'_1 = 0$ works.
- Otherwise let $d'_1 = d' + 1$. Indeed:
  - As $d' \leq d_1 - 1$, we have $d' + 1 \leq d_1 - 1 + 1$ and $d_1 \neq 0$, by lemma 10, $d_1 - 1 + 1 = d_1$, do $d'_1 \leq d_1$.
  - $d' \leq (d' + 1 - 1) \leq (d'_1 - 1)$ and $d' \leq (d_1 - 1) \wedge (x \mapsto d)(y) \leq (x \mapsto d)(y)$ give $d' \leq (d'_1 - 1) \wedge ((x \mapsto d)(y))$.
  - As $d'_1 - 1 = d'$, it follows that $(d'_1 - 1) \wedge ((x \mapsto d)(y)) \leq d'$.

□

Lemma 18 *Let $n \in \mathbb{N}$. If*

$$\gamma' \leq \gamma_0 \wedge \bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j \wedge \bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma_i$$

then there exist γ'_0 and γ'_i for all $i \in \{1 \ldots n\}$, such that $\gamma'_0 \leq \gamma_0$, for all i, $\gamma'_i \leq \gamma_i$, and

$$\gamma' = \gamma'_0 \wedge \bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \wedge \bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma'_i$$

Proof Let $\gamma'_0 = \gamma'$ and for all i, let $\gamma'_i = \gamma_i$. By transitivity we have $\gamma'_0 \leq \gamma_0$ and trivially $\gamma'_i \leq \gamma_i$. We have easily too that

$$\gamma'_0 \wedge \bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \wedge \bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma'_i \leq \gamma'$$

by definition of γ'. Moreover, by hypothesis, we know that $\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \geq \gamma'$ and $\bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma'_i \geq \gamma'$, so

$$\gamma' \leq \gamma'_0 \wedge \bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \wedge \bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma'_i$$

and the expected equality arises. □

C.2 Weakening lemmas

Lemma 19 *If* $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$ *then there exist* γ'_1, γ'_2, d_1, d_2 *such that* $\gamma_1 = \gamma'_1[x \mapsto d_1]$, $\gamma_2 = \gamma'_2[x \mapsto d_2]$, *and* $\gamma = (\gamma'_1 - 1) \wedge d_0 \mathbin{@} \gamma'_2$.

Proof Let $d_1 = \gamma_1(x)$ and $d_2 = \gamma_2(x)$. Let γ'_1 be the function associating $\gamma_1(y)$ to every variable $y \neq x$ and such that $\gamma'_1(x) = \gamma(x) + 1$, which we can write $\gamma_1[x \mapsto \gamma(x) + 1]$. Let γ'_2 be the function associating $\gamma_2(y)$ to every variable $y \neq x$ and such that $\gamma'_2(x) = \infty$, which we can write $\gamma_2[x \mapsto \infty]$. We have trivially $\gamma_1 = \gamma'_1[x \mapsto d_1]$ and $\gamma_2 = \gamma'_2[x \mapsto d_2]$. We now check the third property. On x,

$$\gamma(x) = (\gamma(x) + 1 - 1) \wedge d_0 \mathbin{@} \infty = (\gamma'_1(x) - 1) \wedge d_0 \mathbin{@} \gamma'_2(x)$$

On $y \neq x$,

$$\gamma(y) = (\gamma_1(y) - 1) \wedge d_0 \mathbin{@} \gamma_2(y) = (\gamma'_1(y) - 1) \wedge d_0 \mathbin{@} \gamma'_2(y)$$

□

Lemma 20 *If* $\gamma[x \mapsto d] = \gamma_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j \big) \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i \big)$ *then there exist* γ'_0 *and a* γ'_i *for each* i, *such that* $\gamma'_0[x \mapsto d_0] = \gamma_0$, $\gamma'_i[x \mapsto d'_i] = \gamma_i$, *and* $\gamma = \gamma'_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \big) \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i \big)$, *with* $d_0 = \gamma_0(x)$ *and* $d'_i = \gamma_i(x)$ *for all* i.

Proof Let $\gamma'_0 = \gamma_0[x \mapsto \gamma(x)]$ and $\gamma'_i = \gamma_i[x \mapsto \infty]$ for all i. Similarly, it works. □

D Soundness of the translation

Some notions are defined in section 5.2, they are supposed to be know here.

D.1 Three lemmas on $\boldsymbol{\lambda}_B$

Lemma 21 (single `let rec`) *The following typing rule is admissible for the type system of* λ_B.

$$\frac{\begin{array}{c} \Gamma + \{x \mapsto \tau'\} \vdash M : \tau \;/\; \gamma_1[x \mapsto d] \\ \Gamma + \{x \mapsto \tau'\} \vdash N : \tau' \;/\; \gamma_2[x \mapsto d'] \qquad d' \geq 1 \end{array}}{\Gamma \vdash \textbf{let rec } x = N \textbf{ in } M : \tau \;/\; \gamma_1 \wedge d \mathbin{@} \gamma_2} \text{(singlerec)}$$

Proof Easy. □

Lemma 8 (n abstractions) *If* $\Gamma + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash M : \tau \;/\; (\gamma - n)[\ldots\ x_i \mapsto d_i\ \ldots]$, *then* $\Gamma \vdash \boldsymbol{\lambda}(x_1, \ldots, x_n).M : \tau_1 \xrightarrow{d_1 + (n-1)} \tau_2 \xrightarrow{d_2 + (n-2)} \ldots \tau_n \xrightarrow{d_n} \tau \;/\; \gamma$.

Proof Skipped for now, but should be written, although not too hard. □

Lemma 9 **(n applications)** *If* $\Gamma \vdash M : \tau_1 \xrightarrow{d_1+(n-1)} \tau_2 \xrightarrow{d_2+(n-2)} \dots \tau_n \xrightarrow{d_n} \tau \ / \ \gamma$, *and for all* $i \in \{1 \dots n\}$, $\Gamma(x_i) = \tau_i$, *then* $\Gamma \vdash M(x_1, \dots, x_n) : \tau \ / \ (\gamma - n) \wedge (\dots \ x_i \mapsto d_i \ \dots)$.

D.2 Soundness

We define $\mathit{IsRec}(E)$ by 1 if $E = \lambda x.C$ and 0 otherwise, and extend it to annotated expressions. Before to prove the soundness of the translation, we need a technical lemma, for the `close`() operator.

Lemma 22 **(close)** – *Let* $\mathcal{T} = \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$ *be a well-formed mixin type with* $\mathcal{I}(X) = \mathcal{O}(X)$ *on* $\mathit{dom}(\mathcal{O}) \cap \mathit{dom}(\mathcal{I})$ *and* e *a variable.*

- *Let* Γ *be an environment such that* $\Gamma(e) = \mathcal{T}$.
- *Let* $((X^1_1, \dots, X^1_{n_1}), \dots, (X^p_1, \dots, X^p_{n_p}))$ *be a serialization of* $\mathcal{D}$ *against* $\mathcal{O}$.
- *Let* ι *be an input assignment from fresh* x^j_i*s to* X^j_i*s.*
- *Let* R *be the record*

$$\begin{aligned}\langle\ & X^1_1 = x^1_1; \dots; X^1_{n_1} = x^1_{n_1}; \\ & \dots; \\ & X^p_1 = x^p_1; \dots; X^p_{n_p} = x^p_{n_p} \rangle\end{aligned}$$

- *Let* $\tau = \langle [\![]\!] \circ \mathcal{O} \rangle = \langle X^1_1 : \tau^1_1; \dots; X^1_{n_1}; \dots; X^p_1 : \tau^p_1; \dots; X^p_{n_p} \rangle$, *with* $\tau^i_j = [\![\mathcal{O}(X^i_j)]\!]$ *for all* i, j.
- *For all* $i \in \{1 \dots p\}$, *and for all* $j \in \{1 \dots n_i\}$, *let* $M^i_j = e.X^i_j \ (\iota^{-1}\mathcal{D}^{-1}(X^i_j))$.
- *For all* $i \in \{1 \dots p\}$, *let* B_i *be the binding* $x^i_1 = M^i_1$ **and** $\dots$ **and** $x^i_{n_i} = M^i_{n_i}$.
- *For all* $i \in \{1 \dots p\}$, *let* $M_i =$ **let rec** B_i **in** $\dots$ **let rec** B_p **in** R, *and* $M_{p+1} = R$ *by extension.*

  Then, for all $i \in \{1 \dots p+1\}$,

$$\begin{aligned}&\Gamma + \{\dots\ x^1_j : \tau^1_j \ \dots; \dots; \dots\ x^{i-1}_j : \tau^{i-1}_j \ \dots\} \\ &\quad \vdash M_i : \tau \ / \ d^o(\Gamma) \wedge (x^1_j {\mapsto} 0 \mid 1 \leq j \leq n_1) \wedge \dots \wedge (x^{i-1}_j {\mapsto} 0 \mid 1 \leq j \leq n_{i-1})\end{aligned}$$

Proof We proceed by decreasing induction.

Base $i = p + 1$. The base case is easy, because we have directly

$$\begin{aligned}&\Gamma + \{\dots\ x^1_j : \tau^1_j \ \dots; \dots; \dots\ x^p_j : \tau^p_j \ \dots\} \\ &\quad \vdash R : \tau \ / \ d^o(\Gamma) \wedge (x^1_j {\mapsto} 0 \mid 1 \leq j \leq n_1) \wedge \dots \wedge (x^p_j {\mapsto} 0 \mid 1 \leq j \leq n_p)\end{aligned}$$

Step Let $i \geq 2$, we suppose that our lemma is true for i and try and prove it for $i - 1$. By induction hypothesis, we have

$$\begin{aligned}&\Gamma + \{\dots\ x^1_j : \tau^1_j \ \dots; \dots; \dots\ x^{i-1}_j : \tau^{i-1}_j \ \dots\} \\ &\quad \vdash M_i : \tau \ / \ d^o(\Gamma) \wedge (x^1_j {\mapsto} 0 \mid 1 \leq j \leq n_1) \wedge \dots \wedge (x^{i-1}_j {\mapsto} 0 \mid 1 \leq j \leq n_{i-1})\end{aligned}$$

and $M_{i-1} = \textbf{let rec } B_{i-1} \textbf{ in } M_i$. Naturally, we apply the rule [rec]. The premisse which consists in typing M_i is given by induction hypothesis. Therefore, we just must type M_k^{i-1} for all $k \in \{1 \ldots n_{i-1}\}$:

$$\begin{array}{l} \mathit{\Gamma} + \{\ldots\ x_j^1 : \tau_j^1\ \ \ldots;\ldots;\ldots\ x_j^{i-1} : \tau_j^{i-1}\ \ \ldots\} \\ \quad \vdash M_k^{i-1} : \tau_k^{i-1} \\ \quad /\ d^o(\mathit{\Gamma}) \wedge (x_j^1 \mapsto 0 \mid 1 \leq j \leq n_1) \wedge \ldots \wedge (x_j^{i-2} \mapsto 0 \mid 1 \leq j \leq n_{i-2}) \\ \qquad\qquad \wedge\, (x_j^{i-1} \mapsto 1 \mid 1 \leq j \leq n_{i-1}) \end{array}$$

Let $(Y_1, \ldots, Y_q) = \mathcal{D}^{-1}(X_k^{i-1})$ and $(y_1, \ldots, y_q)\iota^{-1}(\mathcal{D}^{-1}(X_k^{i-1}))$. As the X_j^i are a serialization of $\mathcal{D}$, the Y_l are all X_m^j, with $j \leq i-1$. Let ϵ and η be two maps from $\{1 \ldots q\}$, such that for all $l \in \{1 \ldots q\}$, $Y_l = X_{\eta(l)}^{\epsilon(l)}$ (and $y_l = x_{\eta(l)}^{\epsilon(l)}$).
Therefore, we have for all l

$$(\mathit{\Gamma} + \{\ldots\ x_j^1 : \tau_j^1\ \ \ldots;\ldots;\ldots\ x_j^{i-1} : \tau_j^{i-1}\ \ \ldots\})(y_l) = \tau_{\eta(l)}^{\epsilon(l)}$$

But, what we have too is

$$\begin{array}{l} \mathit{\Gamma} + \{\ldots\ x_j^1 : \tau_j^1\ \ \ldots;\ldots;\ldots\ x_j^{i-1} : \tau_j^{i-1}\ \ \ldots\} \\ \qquad\qquad \vdash e.X_k^{i-1} : [\![\tau_k^{i-1}]\!]_{X_k^{i-1},\mathcal{D},\mathcal{I}}\ /\ d^o(\mathit{\Gamma}) \end{array}$$

with

$$[\![\tau_k^{i-1}]\!]_{X_k^{i-1},\mathcal{D},\mathcal{I}} = \tau_{\eta(1)}^{\epsilon(1)} \xrightarrow{\chi_1+(q-1)} \ldots\ \tau_{\eta(q)}^{\epsilon(q)} \xrightarrow{\chi_q} \tau_k^{i-1}$$

Thus, we can derive

$$\begin{array}{l} \mathit{\Gamma} + \{\ldots\ x_j^1 : \tau_j^1\ \ \ldots;\ldots;\ldots\ x_j^{i-1} : \tau_j^{i-1}\ \ \ldots\} \\ \quad \vdash M_k^{i-1} : \tau_k^{i-1} \\ \quad /\ d^o(\mathit{\Gamma}) \wedge (x_{\eta(l)}^{\epsilon(l)} \mapsto \chi_1 \mid 1 \leq l \leq q) \end{array}$$

But as the X_j^is are a serialization of $\mathcal{D}$, we know that if $\epsilon(l) = i-1$, then as $\mathcal{D}$ is correct, $\chi_l = 1$, so the degree environment proposed is correct:
$d^o(\mathit{\Gamma}) \wedge (x_j^1 \mapsto 0 \mid 1 \leq j \leq n_1) \wedge \ldots \wedge (x_j^{i-2} \mapsto 0 \mid 1 \leq j \leq n_{i-2})$
$\wedge\, (x_j^{i-1} \mapsto 1 \mid 1 \leq j \leq n_{i-1})$.

□

Theorem 2 (soundness of the translation) *If* $\mathit{\Gamma} \vdash E : \mathcal{T}$, *then* $[\![\mathit{\Gamma}]\!] \vdash [\![\overline{E}]\!] : [\![\mathcal{T}]\!]\ /\ d^o(\mathit{\Gamma}) + \mathit{IsRec}(E)$.

Proof

We reason by induction on E. The only difficult cases are structure construction, feezing, and closing.

1. $E = \langle \iota ; o \rangle$ and $\mathcal{T} = \{\mathcal{I};\mathcal{O};\mathcal{D}\}$. By typing, we have $\mathcal{D} = \mathcal{D}\langle \iota ; o\rangle$, $\vdash \mathcal{D}$, $dom(o) = dom(\mathcal{O})$, and for all $X \in dom(o)$, $\Gamma + \mathcal{I} \circ \iota \vdash o(X) : \mathcal{O}(X)$.

   Let $o = X_i \overset{i\in I}{\mapsto} E_i$, $\mathcal{O} = X_i \overset{i\in I}{\mapsto} \mathcal{T}_i$, $\chi_i = \mathit{IsRec}(E_i)$ and $\iota = y_j \overset{j\in J}{\mapsto} Y_j$, with $\mathcal{I}(Y_j) = \mathcal{T}'_j$ for all j, with the X_is and Y_js ordered lexicographically, that is, if $i_1 < i_2$, then $X_{i_1} <_{lex} X_{i_2}$, and similarly for the Y_js.
   By induction hypothesis, for all i, we have $\llbracket\Gamma\rrbracket + \llbracket\mathcal{I}\circ\iota\rrbracket \vdash \llbracket\overline{E_i}\rrbracket : \llbracket\mathcal{T}_i\rrbracket \ / \ d^o(\Gamma + \mathcal{I}\circ\iota) + \chi_i$.
   But $FV(\llbracket\overline{E_i}\rrbracket) = FV(E_i)$ and $FV(E_i) \cap dom(\iota) = \iota^{-1}(\mathcal{D}^{-1}(X_i))$, so we can apply lemma 8, and weakening lemmas 12 and 13 to eliminate variables of $dom(\iota)$ that are not free in E_i. Let $(Z_1, \ldots, Z_n) = \mathcal{D}^{-1}(X_i)$ and for all $k \in \{1 \ldots n\}$, $\mathcal{T}''_k = \mathcal{I}(Z_k)$. We obtain

   $$\Gamma \vdash \boldsymbol{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X_i)).\llbracket\overline{E_i}\rrbracket : \llbracket\mathcal{T}''_1\rrbracket \xrightarrow{\chi_i + (n-1)} \ldots \llbracket\mathcal{T}''_n\rrbracket \xrightarrow{\chi_i} \llbracket\mathcal{T}_i\rrbracket \ / \ d^o(\Gamma)$$

   But $\llbracket\mathcal{T}_i\rrbracket_{X_i,\mathcal{D},\mathcal{I}} = \llbracket\mathcal{T}''_1\rrbracket \xrightarrow{\chi_i+(n-1)} \ldots \llbracket\mathcal{T}''_n\rrbracket \xrightarrow{\chi_i} \llbracket\mathcal{T}_i\rrbracket$, because $\mathcal{D}(Z_k, X_i) = \nu(\iota^{-1}(Z_k),\ E_i) = \mathit{IsRec}(E_i) = \chi_i$. So

   $$\llbracket\Gamma\rrbracket \vdash \langle \ldots \ X_i = \boldsymbol{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X_i)).\llbracket\overline{E_i}\rrbracket \ \ldots\rangle : \langle \ldots \ X_i : \llbracket\mathcal{T}_i\rrbracket_{X_i,\mathcal{D},\mathcal{I}}\rangle \ / \ d^o(\Gamma)$$

2. $E = E_1 \,!\, X$, $\overline{E} = \overline{E_1} \,!\, X$. By typing, we have
   - $\Gamma \vdash E : \{\mathcal{I}_{\setminus X};\mathcal{O};\mathcal{D}!X\}$
   - $\Gamma \vdash E_1 : \{\mathcal{I};\mathcal{O};\mathcal{D}\}$

   Let $\mathcal{D}' = \mathcal{D}!X$, $\mathcal{T}' = \{\mathcal{I}_{\setminus X};\mathcal{O};\mathcal{D}!X\}$, and $\mathcal{T} = \{\mathcal{I};\mathcal{O};\mathcal{D}\}$. Fix a ι covering the names mentioned in $\mathcal{I}$ and $\mathcal{O}$. By induction hypothesis, we know that

   $$\llbracket\Gamma\rrbracket \vdash \llbracket\overline{E_1}\rrbracket : \langle Y : \llbracket\mathcal{O}(Y)\rrbracket_{Y,\mathcal{D},\mathcal{I}} \mid Y \in dom(\mathcal{O})\rangle \ / \ d^o(\Gamma)$$

   But

   $$\begin{aligned}\llbracket\overline{E}\rrbracket = \textbf{let } & e = \llbracket\overline{E_1}\rrbracket \textbf{ in}\\ & \langle Z = e.Z \ \mid Z \in dom(\mathcal{O}), X \notin \mathcal{D}^{-1}(Z);\\ & \ Y = \boldsymbol{\lambda}\iota^{-1}({\mathcal{D}'}^{-1}(Y)).\textbf{let rec } \iota^{-1}(X) = e.X \ \iota^{-1}(\mathcal{D}^{-1}(X)) \textbf{ in}\\ & \qquad\qquad e.Y \iota^{-1}(\mathcal{D}^{-1}(Y)) \mid X \in \mathcal{D}^{-1}(Y)\rangle\end{aligned}$$

   and

   $$\llbracket\mathcal{T}''\rrbracket = \langle Y : \llbracket\mathcal{O}(Y)\rrbracket_{Y,\mathcal{D}!X,\mathcal{I}_{\setminus X}} \mid Y \in dom(\mathcal{O})\rangle$$

   Notice that the condition $X \in \mathcal{D}^{-1}(Y)$ in $\llbracket\overline{E}\rrbracket$, implies that $Y \in dom(\mathcal{O})$ because $\mathcal{T}$ is assumed to be well-formed. So we must check that for all $Y \in dom(\mathcal{O})$,

   $$\llbracket\Gamma + \{e \mapsto \mathcal{T}\}\rrbracket \vdash \ldots : \llbracket\mathcal{O}(Y)\rrbracket_{Y,\mathcal{D}!X,\mathcal{I}_{\setminus X}} \ / \ d^o(\Gamma)[e \mapsto 0]$$

   where the $\ldots$ vary according to whether $X \in \mathcal{D}^{-1}(Y)$ or not.
   - $X \notin \mathcal{D}^{-1}(Y)$. Let

     $$\llbracket\mathcal{O}(Y)\rrbracket_{Y,\mathcal{D}!X,\mathcal{I}_{\setminus X}} = \llbracket\mathcal{T}_1\rrbracket \xrightarrow{\chi_1+(n-1)} \ldots \llbracket\mathcal{T}_n\rrbracket \xrightarrow{\chi_n} \llbracket\mathcal{O}(Y)\rrbracket$$

     where

  - $(\mathcal{T}_1^{\chi_1}, \ldots, \mathcal{T}_n^{\chi_n}) = FCT_{\mathcal{D}}(Y, \mathcal{I})$,
  - $(\mathcal{D}!X)^{-1}(Y) = (X_1, \ldots, X_n)$,
  - and for all $i \in \{1 \ldots n\}$, $\mathcal{T}_i = \mathcal{I}_{\setminus X}(X_i)$ and $\mathcal{D}'(X_i, Y) = \chi_i$.

  But $\mathcal{D}' = \mathcal{D}!X = (\mathcal{D} \cup \mathcal{D}_{around}) \setminus \mathcal{D}_{remove}$, where $\mathcal{D}_{around} = \{Y' \xrightarrow{\chi'_1 \wedge \chi'_2} Z' \mid Y' \xrightarrow{\chi'_1} X \text{ and } X \xrightarrow{\chi'_2} Z'\}$) and $\mathcal{D}_{remove} = \{X \xrightarrow{\chi'} Z'\}$, so $\mathcal{D}'^{-1}(Y) = \mathcal{D}^{-1}(Y)$, because

  - as $X \notin \mathcal{D}^{-1}(Y)$, there is no edge of the form $X \xrightarrow{\chi'} Y$ in $\mathcal{D}$, so the $\mathcal{D}_{remove}$ does not remove any edge leading to Y,
  - and moreover, as there is no edge of the form $X \xrightarrow{\chi'} Y$ in $\mathcal{D}$, the $\mathcal{D}_{around}$ does not introduce new edges leading to Y either.

  Additionally, $\mathcal{D}'(X_i, Y) = \mathcal{D}(X_i, Y)$, so that

  $$[\![\mathcal{O}(Y)]\!]_{Y,\mathcal{D}!X,\mathcal{I}_{\setminus X}} = [\![\mathcal{O}(Y)]\!]_{Y,\mathcal{D},\mathcal{I}}$$

  which is clearly the type of $e.Y$.

- $X \in \mathcal{D}^{-1}(Y)$ Let
  - $M_1 = e.Y\ \iota^{-1}(\mathcal{D}^{-1}(Y))$
  - $M_2 = e.X\ \iota^{-1}(\mathcal{D}^{-1}(X))$
  - $M_3 =$ **let rec** $x = M_2$ **in** M_1
  - $M_4 = \boldsymbol{\lambda}\iota^{-1}(\mathcal{D}'^{-1}(Y)).M_3$
  - $\Gamma' = [\![\Gamma + \{e \mapsto \mathcal{T}\}]\!]$
  - $\gamma = d^o(\Gamma)[e \mapsto 0]$

  Let $\chi = \mathcal{D}(X, Y)$, $x = \iota^{-1}(X)$ (by ι defined above), and, about the predecessors of Y in $\mathcal{D}'$:
  - $(X_1, \ldots, X_n) = \mathcal{D}'^{-1}(Y)$
  - $(x_1, \ldots, x_n) = \iota^{-1}(\mathcal{D}'^{-1}(Y))$
  - $(\chi_1, \ldots, \chi_n) = (\mathcal{D}'(X_1, Y), \ldots, \mathcal{D}'(X_n, Y))$
  - $(\mathcal{T}_1, \ldots, \mathcal{T}_n) = (\mathcal{I}(X_1), \ldots, \mathcal{I}(X_n))$

  We know that X is one of the X'_js, say X'_{j_0}, with $x = x'_{j_0}$ and $\chi = \chi'_{j_0}$. About the predecessors of Y in $\mathcal{D}$:
  - $(X'_1, \ldots, X'_m) = \mathcal{D}^{-1}(Y)$
  - $(x'_1, \ldots, x'_m) = \iota^{-1}(\mathcal{D}^{-1}(Y))$
  - $(\chi'_1, \ldots, \chi'_m) = (\mathcal{D}(X'_1, Y), \ldots, \mathcal{D}(X'_m, Y))$
  - $(\mathcal{T}'_1, \ldots, \mathcal{T}'_m) = (\mathcal{I}(X'_1), \ldots, \mathcal{I}(X'_m))$

  And about the predecessors of X in $\mathcal{D}$:
  - $(X''_1, \ldots, X''_p) = \mathcal{D}^{-1}(X)$
  - $(x''_1, \ldots, x''_p) = \iota^{-1}(\mathcal{D}^{-1}(X))$
  - $(\chi''_1, \ldots, \chi''_p) = (\mathcal{D}(X''_1, X), \ldots, \mathcal{D}(X''_p, X))$
  - $(\mathcal{T}''_1, \ldots, \mathcal{T}''_p) = (\mathcal{I}(X''_1), \ldots, \mathcal{I}(X''_p))$

  We want to derive the following, with $\Gamma'' = \Gamma' + \{\ldots\ x_i : [\![\mathcal{T}_i]\!]\ \ldots\} + \{x \mapsto [\![\mathcal{O}(X)]\!]\}$

  $$\cfrac{\cfrac{\begin{array}{c}\Gamma'' \vdash M_1 : [\![\mathcal{O}(Y)]\!] \ / \ d^o(\Gamma')[\ldots\ x'_j \mapsto \chi'_j\ \ldots] \\ \Gamma'' \vdash M_2 : [\![\mathcal{O}(X)]\!] \ / \ d^o(\Gamma')[\ldots\ x''_k \mapsto \chi''_k\ \ldots]\end{array}}{\Gamma' + \{\ldots\ x_i : [\![\mathcal{T}_i]\!]\ \ldots\} \vdash M_3 : [\![\mathcal{O}(Y)]\!] \ / \ \gamma[\ldots\ x_i \mapsto \chi_i\ \ldots]}}{\Gamma' \vdash M_4 : [\![\mathcal{O}(Y)]\!]_{Y,\mathcal{D}',\mathcal{I}} \ / \ \gamma}$$

For this, we must

(a) Derive the two premisses.

(b) Prove that it is a correct derivation with respect to the [rec] and [abstr] rules.

(a) For the two premisses, the lemma 9 does the job well.

(b) At the level of the type enivronments, it is easy, as well as for the terms and types. The difficulty is in checking that degree environments follow the patterns described by the rules. For the n abstraction rules, it is not difficult. For the [rec] one, we have

$$\begin{aligned}
&\{X_i \xrightarrow{\chi_i} Y \mid i \in \{1 \dots n\}\} = \\
&\quad (\{X'_j \xrightarrow{\chi'_j} Y \mid j \in \{1 \dots m\}\} \\
&\quad \cup \{X'_k \xrightarrow{\chi \wedge \chi''_k} Y \mid k \in \{1 \dots p\}\}) \\
&\setminus \{X \xrightarrow{\chi} Y\}
\end{aligned} \tag{1}$$

by construction of $\mathcal{D}'$. So every x_i is either an x'_j, or an x''_k, or both (but is different from x at the same time). So, to prove that our derivation is correct, we just have to prove that for all i,

$$\dots\ x_i \mapsto \chi_i\ \dots = (\dots\ x'_j \mapsto \chi'_j\ \dots) \wedge \chi \mathbin{@} (\dots\ x''_k \mapsto \chi''_k\ \dots)$$

- If $x_i \neq x'_j$ for all j, $x_i = x''_k$, $x''_k \neq x$. By 1, $\chi_i = \chi \mathbin{@} \chi''_k = ((\dots\ x'_j \mapsto \chi'_j\ \dots) \wedge \chi \mathbin{@} (\dots\ x''_k \mapsto \chi''_k\ \dots))(x_i)$.
- If $x_i \neq x''_k$ for all k, $x_i = x'_j$, $x'_j \neq x$. By 1, $\chi_i = \chi'_j = ((\dots\ x'_j \mapsto \chi'_j\ \dots) \wedge \chi \mathbin{@} (\dots\ x''_k \mapsto \chi''_k\ \dots))(x_i)$.
- If $x_i = x'_j = x''_k \neq x$, then by 1, χ_i is the minimum valuation of the edges from X_i to Y, so $\chi_i = \chi'_j \wedge \chi \mathbin{@} \chi''_k = ((\dots\ x'_j \mapsto \chi'_j\ \dots) \wedge \chi \mathbin{@} (\dots\ x''_k \mapsto \chi''_k\ \dots))(x_i)$.

So it works.

3. $E = \texttt{close}(E_1)$ The lemma 22 does the job.

□

Values

$v ::= x \mid \lambda x.M \mid \langle \dots\ X_i = v_i\ \dots \rangle \mid c$

Evaluation contexts

$$\begin{aligned}
\mathbb{E} ::= {} & [\,]\ M \mid v\ [\,] \mid [\,].X \\
& \mid \textbf{let rec}\ \dots\ x_{i-1} = v_{i-1}\ \textbf{and}\ x_i = [\,]\ \textbf{and}\ \dots\ x_n = M_n\ \textbf{in}\ M \\
& \mid \textbf{let}\ x = [\,]\ \textbf{in}\ M \\
& \mid \langle \dots ; X_{i-1} = v_{i-1} ; X_i = [\,] ; X_{i+1} = M_{i+1} ; \dots \rangle
\end{aligned}$$

Parallel substitution

Let $\rho = \dots\ x_i \leftarrow M_i\ \dots$, we define

$$\begin{array}{rcll}
x\{\rho\} & = & M_i & \text{if } x = x_i \\
x\{\rho\} & = & x & \text{otherwise} \\
(\lambda x.M)\{\rho\} & = & \lambda x.(M\{\rho\}) & \text{if } x \notin \bigcup_i (\{x_i\} \cup FV(M_i)) \\
(M_1\ M_2)\{\rho\} & = & M_1\{\rho\}\ M_2\{\rho\} & \\
(\textbf{let rec}\ \dots\ y_k = N_k\ \dots\ \textbf{in}\ M)\{\rho\} & = & \multicolumn{2}{l}{\textbf{let rec}\ \dots\ y_k = N_k\{\rho\}\ \dots\ \textbf{in}\ M\{\rho\}} \\
 & & \multicolumn{2}{r}{\text{if } (\bigcup_k \{y_k\}) \cap \bigcup_i (\{x_i\} \cup FV(M_i)) \neq \emptyset}
\end{array}$$

Rules for records easy

Reduction rules

$$\begin{array}{rcll}
(\lambda x.M)\ v & \to & M\{x \leftarrow v\} & \text{(beta)} \\
\textbf{let rec}\ \dots\ x_i = v_i\ \dots\ \textbf{in}\ M & \to & M\{\dots\ x_i \leftarrow M_i\ \dots\} & \text{(mutrec)} \\
\langle \dots\ X_i = v_i\ \dots \rangle.X_{i_0} & \to & v_{i_0} & \text{(select)} \\
\textbf{let}\ x = v\ \textbf{in}\ M & \to & M\{x \leftarrow v\} & \text{(bind)} \\
\multicolumn{4}{c}{\text{with for all } j,\ M_j = \textbf{let rec}\ \dots\ x_i = v_i\ \dots\ \textbf{in}\ v_j}
\end{array}$$

$$\frac{M \to M'}{\mathbb{E}\,[M] \to \mathbb{E}\,[M']}\ \text{(context)}$$

Fig. 6. Dynamic semantics of λ_B

$$\llbracket (E : \mathcal{T}').X : \mathcal{T} \rrbracket = \llbracket E : \mathcal{T}' \rrbracket.X$$

$$\llbracket \langle \iota; o \rangle : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \rrbracket =$$

$$\quad \langle X = \boldsymbol{\lambda} \iota^{-1}(\mathcal{D}^{-1}(X)).\llbracket o(X) : \mathcal{O}(X) \rrbracket \mid X \in dom(\mathcal{O}) \rangle$$

$$\llbracket (E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\}) + (E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\}) : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \rrbracket =$$

$$\quad \mathbf{let}\ e_1 = \llbracket E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\} \rrbracket\ \mathbf{in}\ \mathbf{let}\ e_2 = \llbracket E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\} \rrbracket\ \mathbf{in}$$

$$\quad \langle X = e_1.X \mid X \in dom(\mathcal{O}_1);$$

$$\quad\ \ Y = e_2.Y \mid Y \in dom(\mathcal{O}_2) \rangle$$

$$\llbracket (E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) \setminus X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \rrbracket =$$

$$\quad \mathbf{let}\ e = \llbracket E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\} \rrbracket\ \mathbf{in}\ \langle Y = e.Y \mid Y \in dom(\mathcal{O}) \rangle$$

$$\llbracket (E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})[X \leftarrow Y] : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \rrbracket =$$

$$\quad \mathbf{let}\ e = \llbracket E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\} \rrbracket\ \mathbf{in}$$

$$\quad \langle Z\{X \leftarrow Y\} = \boldsymbol{\lambda} \overline{\mathcal{D}^{-1}(Z\{X \leftarrow Y\})}.(e.Z\ \overline{\mathcal{D}'^{-1}(Z)})\{\overline{X} \leftarrow \overline{Y}\} \mid Z \in dom(\mathcal{O}') \rangle$$

$$\llbracket (E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})\, !\, X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\} \rrbracket =$$

$$\quad \mathbf{let}\ e = \llbracket E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\} \rrbracket\ \mathbf{in}$$

$$\quad \langle Z = e.Z \mid Z \in dom(\mathcal{O}),\ X \notin \mathcal{D}'^{-1}(Z);$$

$$\quad\ \ Y = \boldsymbol{\lambda} \overline{\mathcal{D}^{-1}(Y)}.\mathbf{let\ rec}\ \overline{X} = e.X\ \overline{\mathcal{D}'^{-1}(X)}\ \mathbf{in}\ e.Y\ \overline{\mathcal{D}'^{-1}(Y)} \mid X \in \mathcal{D}'^{-1}(Y) \rangle$$

$$\llbracket \mathtt{close}(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) : \{\emptyset; \mathcal{O}; \emptyset\} \rrbracket =$$

$$\quad \mathbf{let}\ e = \llbracket E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\} \rrbracket\ \mathbf{in}$$

$$\quad \mathbf{let\ rec}\ \overline{X^1_1} = e.X^1_1\ \overline{\mathcal{D}'^{-1}(X^1_1)}\ \mathbf{and}\ \ldots\ \mathbf{and}\ \overline{X^1_{n_1}} = e.X^1_{n_1}\ \overline{\mathcal{D}'^{-1}(X^1_{n_1})}\ \mathbf{in}$$

$$\quad \ldots$$

$$\quad \mathbf{let\ rec}\ \overline{X^p_1} = e.X^p_1\ \overline{\mathcal{D}'^{-1}(X^p_1)}\ \mathbf{and}\ \ldots\ \mathbf{and}\ \overline{X^p_{n_p}} = e.X^p_{n_p}\ \overline{\mathcal{D}'^{-1}(X^p_{n_p})}\ \mathbf{in}$$

$$\quad \langle X = \overline{X} \mid X \in dom(\mathcal{O}) \rangle$$

$$\quad \text{where } (\{X^1_1 \ldots X^1_{n_1}\}, \ldots, \{X^p_1 \ldots X^p_{n_p}\}) \text{ is a serialization of } dom(\mathcal{O}') \text{ against } \mathcal{D}'$$

Fig. 7. The translation scheme

$$\frac{\gamma(x) = 0}{\Gamma \vdash x : \Gamma(x) \ / \ \gamma} \text{ (var)} \qquad\qquad \Gamma \vdash c : TC(c) \ / \ \gamma \text{ (const)}$$

$$\frac{\Gamma + \{x : \tau'\} \vdash M : \tau \ / \ (\gamma - 1)[x \mapsto d]}{\Gamma \vdash \lambda x.M : \tau' \xrightarrow{d} \tau \ / \ \gamma} \text{ (abstr)}$$

$$\frac{\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \ / \ \gamma_1 \qquad \Gamma \vdash M_2 : \tau' \ / \ \gamma_2}{\Gamma \vdash M_1 \ M_2 : \tau \ / \ (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2} \text{ (app)}$$

$$\frac{\Gamma \vdash M : \tau' \xrightarrow{d} \tau \ / \ \gamma \qquad \Gamma(x) = \tau'}{\Gamma \vdash M \ x : \tau \ / \ (\gamma - 1) \wedge (x \mapsto d)} \text{ (appvar)}$$

$$\frac{\Gamma \vdash M : \tau' \ / \ \gamma' \qquad \Gamma + \{x : \tau'\} \vdash N : \tau \ / \ \gamma[x \mapsto d]}{\Gamma \vdash \mathbf{let} \ x = M \ \mathbf{in} \ N : \tau \ / \ \gamma \wedge d \mathbin{@} \gamma'} \text{ (let)}$$

$$\frac{\begin{array}{c} \Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash M : \tau \ / \ \gamma[\ldots \ x_j \mapsto d_j \ \ldots] \\ \forall i : \Gamma + \{\ldots \ x_j : \tau_j \ \ldots\} \vdash M_i : \tau_i \ / \ \gamma_i[\ldots \ x_j \mapsto d_{ij} \ \ldots] \\ \forall i, j : d_{ij} \geq 1 \qquad \forall i, j, k : d_{ik} \leq d_{ij} \mathbin{@} d_{jk} \end{array}}{\Gamma \vdash \mathbf{let\ rec} \ \ldots \ x_i = M_i \ \ldots \ \mathbf{in} \ M : \tau \ / \ \gamma \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)} \text{ (rec)}$$

$$\frac{\forall i : \Gamma \vdash M_i : \tau_i \ / \ \gamma}{\Gamma \vdash \langle \ldots \ X_i = M_i \ \ldots \rangle : \langle \ldots \ X_i : \tau_i \ \ldots \rangle \ / \ \gamma} \text{ (record)}$$

$$\frac{\Gamma \vdash M : \langle \ldots \ X_j : \tau_j \ \ldots \rangle \ / \ \gamma \qquad 1 \leq i \leq n}{\Gamma \vdash M.X_i : \tau_i \ / \ \gamma} \text{ (sel)}$$

Fig. 8. Typing rules for λ_B

- $\mathcal{D}^{-1}(X) = (X_1, \ldots, X_n)$ is the list of the predecessors of X in $\mathcal{D}$, ordered lexicographically.
- $\mathcal{D}(X, Y) = \min \{\chi \mid X \xrightarrow{\chi} Y \in \mathcal{D}\}$
- $FCT_{\mathcal{D}}(X, \mathcal{I}) = (\mathcal{T}_1^{\chi_1}, \ldots, \mathcal{T}_n^{\chi_n})$, for $Pred(\mathcal{D}) \subset dom(\mathcal{I})$, where
    - $\mathcal{D}^{-1}(X) = (X_1, \ldots, X_n)$
    - for all $i \in \{1 \ldots n\}$, $\mathcal{I}(X_i) = \mathcal{T}_i$ and $\mathcal{D}(X_i, X) = \chi_i$.
- $Pred(\mathcal{D}) = \{X \mid X \xrightarrow{\chi} Y \in \mathcal{D}, X, Y \in Names, \chi \in Vals\}$
- $Succ(\mathcal{D}) = \{Y \mid X \xrightarrow{\chi} Y \in \mathcal{D}, X, Y \in Names, \chi \in Vals\}$

Fig. 9. Functions on graphs

Core

$$\begin{aligned}
[\![\tau_1 \to \tau_2]\!] &= \tau_1 \xrightarrow{0} \tau_2\\
[\![\texttt{int}]\!] &= \texttt{int}\\
[\![\texttt{bool}]\!] &= \texttt{bool}
\end{aligned}$$

Mixin

If $\vdash \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$,

$[\![\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \langle X : [\![\mathcal{O}(X)]\!]_{X,\mathcal{D},\mathcal{I}} \mid X \in \mathit{dom}(\mathcal{O})\rangle$

with $[\![\mathcal{O}(X)]\!]_{X,\mathcal{D},\mathcal{I}} = [\![\mathcal{T}_1]\!] \xrightarrow{\chi_1 + (n-1)} [\![\mathcal{T}_2]\!] \xrightarrow{\chi_2 + (n-2)} \ldots\ [\![\mathcal{T}_n]\!] \xrightarrow{\chi_n} [\![\mathcal{T}]\!]$,

where $(\mathcal{T}_1^{\chi_1}, \ldots, \mathcal{T}_n^{\chi_n}) = \mathit{FCT}_{\mathcal{D}}(X, \mathcal{I})$,

and $\mathit{Pred}(\mathcal{D}) \subset \mathit{dom}(\mathcal{I})$, which holds if $\vdash \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$.

Fig. 10. Translation of types

Core terms:	$\overline{C} ::= x^{\mathcal{T}} \mid \mathit{cst}^{\mathcal{T}}$	variables, constants
	$\mid \lambda x.\overline{C}^{\mathcal{T}} \mid (\overline{C}_1 \overline{C}_2)^{\mathcal{T}}$	abstraction, application
	$\mid \overline{E}.X^{\mathcal{T}}$	component projection
Mixin terms:	$\overline{E} ::= \overline{C}$	core term
	$\mid \langle \iota; \overline{o}\rangle^{\mathcal{T}}$	mixin structure
	$\mid (\overline{E}_1 + \overline{E}_2)^{\mathcal{T}}$	sum
	$\mid (\overline{E}[X \leftarrow Y])^{\mathcal{T}}$	rename X to Y
	$\mid (\overline{E} \,!\, X)^{\mathcal{T}}$	freeze X
	$\mid (\overline{E} \setminus X)^{\mathcal{T}}$	delete X
	$\mid (\texttt{close}(\overline{E}))^{\mathcal{T}}$	close
Output assignments:	$\overline{o} ::= X_i \overset{i \in I}{\mapsto} \overline{E}_i$	

Fig. 11. Annotated syntax

Degrees

$$d ::= n \mid \infty$$

Minimum

$$\begin{aligned}
d \wedge \infty &= d\\
\infty \wedge d &= d\\
m \wedge n &= m \wedge_{\mathbb{N}} n
\end{aligned}$$

Composition

$$\begin{aligned}
d \mathbin{@} \infty &= \infty\\
d \mathbin{@} 0 &= 0\\
d \mathbin{@} n &= d
\end{aligned}$$

Plus

$$\begin{aligned}
\infty + n &= \infty\\
m + n &= m +_{\mathbb{N}} n
\end{aligned}$$

Minus

$$\begin{aligned}
\infty - n &= \infty\\
m - n &= m -_{\mathbb{N}} n && \text{if } m \geq n\\
m - n &= 0 && \text{if } m < n
\end{aligned}$$

Fig. 12. Summary of degree operations