Mixin modules in a call-by-value setting

TOM HIRSCHOWITZ
ENS Lyon
and
XAVIER LEROY
INRIA Rocquencourt

The ML module system provides powerful parameterization facilities, but lacks the ability to split mutually recursive definitions across modules and provides insufficient support for incremental programming. A promising approach to solve these issues is Ancona and Zucca's mixin module 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 otherwise 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 translates mixin modules down to a call-by-value λ-calculus extended with a non-standard `let rec` construct.

Categories and Subject Descriptors: D.3.1 [**Programming languages**]: Formal definitions and theory—*Semantics*; D.3.2 [**Programming languages**]: Language classifications—*Applicative (functional) languages*; *object-oriented languages*; D.3.3 [**Programming languages**]: Language constructs and features—*Inheritance*; *modules*; *packages*; *recursion*; F.3.2 [**Logics and meanings of programs**]: Semantics of programming languages—*Operational semantics*; F.3.3 [**Logics and meanings of programs**]: Studies of program constructs—*Type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Mixins, modules, recursion, type systems

1. INTRODUCTION

Modular programming and code reuse are 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 refinement 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);

Authors' addresses: T. Hirschowitz, Laboratoire de l'Informatique du Parallélisme, UMR CNRS–INRIA–ENSL–UCBL 5668, École Normale Supérieure de Lyon, 46, allée d'Italie, 69364 Lyon Cedex 07, France; e-mail: Tom.Hirschowitz@ens-lyon.fr. X. Leroy, INRIA Rocquencourt, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France; e-mail: Xavier.Leroy@inria.fr.

conventional linkers, as well as linking calculi [Cardelli 1997], 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 [Bracha and Cook 1990], then extended to a family of module systems [Duggan and Sourelis 1996; Ancona and Zucca 2002; Flatt and Felleisen 1998; Wells and Vestergaard 2000], 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 [2002] 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 the *CMS* calculus [Ancona and Zucca 2002] 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 sum. 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 if the value of `x` or `y` is needed. This issue was not present in the original concept of mixin, which allowed only syntactic values as mixin components. We call mixins with arbitrary components *mixin modules*, hereafter simply referred to as *mixins* when there is no ambiguity.

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 approach 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 [Cardelli 1997; Leroy 1994] 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 sum 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 [2003]. 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 Boudol [2003], 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 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. Detailed proofs are provided in appendix.

2. OVERVIEW

2.1 The *CMS* calculus of mixins

We start this paper by an overview of the *CMS* module calculus of Ancona and Zucca [2002], using an ML-like syntax for readability. A basic mixin 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 consists of defined components, `let`-bound to an expression, and deferred components, declared but not yet defined. The fundamental operator on mixins 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 > 0 and 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 > 0 and 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 > 0 and 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) + (mix 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 > 0 and 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 "!" operator (pronounced "freeze"). For instance, `Nat ! odd` is equivalent to

```
mix
  let even = let odd = λx. x > 0 and even(x-1) in
             λx. x = 0 or odd(x-1)
  let odd  = λx. x > 0 and 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, to ensure that they do not need to trigger any computations.

A component of a mixin can itself be a mixin. Not only does this provide ML-style nested mixins, but it also supports a general encoding of ML functors [Ancona and Zucca 1999]. 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
```

This encoding extends to curried and higher-order functors. For instance, the curried functor

```
module G = functor (X : S) -> functor (Y : S') -> struct ... end
```

is encoded as follows:

```
mixin G = mix
  ? mixin Arg : S
  mixin X = Arg
  mixin Res = mix
    ? mixin Arg : S'
    mixin Y = Arg
    ...
  end
end
```

In the latter example, the need for the additional bindings `X = Arg` and `Y = Arg` becomes clear: the formal parameter of a functor must be bound to a fixed, conventional name (here `Arg`) so that clients of the functor can apply it without knowing the name of its formal parameter; at the same time, a functor body (the `...` in the example above) may need to refer to several functor parameters, requiring them to have distinct, α-convertible names. A similar trick is used to encode the λ-calculus into the ς-calculus of Abadi and Cardelli [1996].

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 mixins, 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 are. Then, the ill-founded recursion cannot be detected. This is the major problem we face in extending ML with mixins.

A partial solution to this problem is to detect ill-founded recursions at execution time, and generate a run-time error. This can be achieved by lazy evaluation of the right-hand sides of recursive definitions. Operationally, to evaluate a recursive definition $x_1 = e_1$ **and** ... **and** $x_n = e_n$, each x_i is bound to a thunk for e_i; these thunks are then evaluated in sequence, memoizing their values; if the evaluation of e_i needs the value of x_j and the thunk e_j is not yet computed, its evaluation is performed and memoized at that time; finally, the ill-founded case where the evaluation of e_i requires its own value via a reference to x_i is detected and reported as a run-time error. This approach is used for evaluating recursive modules in Moscow ML [Russo 2001]. A simplification of this approach is used to evaluate the `letrec` construct of Scheme: the recursively-defined variables x_i are initialized with a special "do not use" value; the right-hand sides e_i are evaluated in sequence, raising an error if a variable evaluates to the "do not use" value; and finally the initial variable values are updated in place with the values of the right-hand sides. While practical and easy to implement, these approaches have the drawback that ill-founded recursive definitions (as in the `Bad` example above) are not detected until run-time. To increase program safety, we would much prefer to detect ill-founded definitions statically, at compile time.

To achieve this goal, our approach consists in enriching mixin signatures with graphs representing the dependencies between components of a mixin, 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:

$$\texttt{Nat:}\quad \begin{tikzcd} \texttt{even} \arrow[r, bend right, "1"'] & \texttt{odd} \arrow[l, bend right, "1"'] \end{tikzcd} \qquad\qquad \texttt{Bad:}\quad \begin{tikzcd} \texttt{x} \arrow[r, bend right, "0"'] & \texttt{y} \arrow[l, bend right, "0"'] \end{tikzcd}$$

An edge $X \xrightarrow{\chi} Y$ expresses that X is used by the definition of Y. 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. (This situation typically arises when a module involved in a mutually recursive definition needs to perform initializing computations.)

```
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
```

Core terms:	$C ::= x \mid cst$	variable, constant
	$\mid \lambda x.C \mid C_1\ C_2$	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
	$\mid \texttt{close}(E)$	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:	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau \to \tau$	
Mixin types:	$\mathcal{T} ::= \tau$	core type
	$\mid \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$	mixin signature
Type assignments:	$\mathcal{I}, \mathcal{O} ::= X_i \overset{i \in I}{\mapsto} \mathcal{T}_i$	
Dependency graphs:	$\mathcal{D}$ (see section 3.2)	

Fig. 1. Syntax of CMS_v

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

$$\begin{tikzcd} \mathtt{u} & \mathtt{f} \arrow[l, "0"'] \arrow[r, bend left, "1"] & \mathtt{g} \arrow[l, bend left, "1"] \arrow[r, "0"] & \mathtt{v} \end{tikzcd}$$

It satisfies the correctness criterion, thus this definition is accepted. Other systems that record a global "valuability" flag on each signature, such as the recursive modules of [Crary et al. 1999], 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 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.

A mixin structure $\langle \iota; o \rangle$ is composed of an *input assignment* ι and an *output*

Free variables:

$$
\begin{array}{rclrcl}
FV(x) & = & \{x\} & FV(cst) & = & \emptyset \\
FV(\lambda x.C) & = & FV(C) \setminus \{x\} & FV(C_1\ C_2) & = & FV(C_1) \cup FV(C_2) \\
FV(\langle \iota; o \rangle) & = & FV(o) \setminus dom(\iota) & FV(X_i \overset{i \in I}{\mapsto} E_i) & = & \bigcup_{i \in I} FV(E_i) \\
FV(E_1 + E_2) & = & FV(E_1) \cup FV(E_2) & FV(E\,!\,X) & = & FV(E) \\
FV(E.X) & = & FV(E) & FV(E[X \leftarrow Y]) & = & FV(E) \\
FV(E \setminus X) & = & FV(E) & FV(\mathtt{close}(E)) & = & FV(E)
\end{array}
$$

Substitution:

$$
\begin{array}{rcl}
y\{x \leftarrow E\} & = & E \text{ if } y = x,\ y \text{ otherwise} \\
cst\{x \leftarrow E\} & = & cst \\
\lambda y.C\{x \leftarrow E\} & = & \lambda y.C\{x \leftarrow E\} \text{ if } y \notin FV(E) \cup \{x\} \\
(C_1\ C_2)\{x \leftarrow E\} & = & C_1\{x \leftarrow E\}\ C_2\{x \leftarrow E\} \\
\langle \iota; o \rangle\{x \leftarrow E\} & = & \langle \iota; o\{x \leftarrow E\} \rangle \text{ if } x \notin dom(\iota) \\
(X_i \overset{i \in I}{\mapsto} E_i)\{x \leftarrow E\} & = & X_i \overset{i \in I}{\mapsto} E_i\{x \leftarrow E\} \\
(E_1 + E_2)\{x \leftarrow E\} & = & E_1\{x \leftarrow E\} + E_2\{x \leftarrow E\} \\
E'[X \leftarrow Y]\{x \leftarrow E\} & = & E'\{x \leftarrow E\}[X \leftarrow Y] \\
E' \setminus X\{x \leftarrow E\} & = & E'\{x \leftarrow E\} \setminus X \\
E'\,!\,X\{x \leftarrow E\} & = & E'\{x \leftarrow E\}\,!\,X \\
E'.X\{x \leftarrow E\} & = & E'\{x \leftarrow E\}.X \\
\mathtt{close}(E')\{x \leftarrow E\} & = & \mathtt{close}(E'\{x \leftarrow E\})
\end{array}
$$

Fig. 2. Operations on CMS_v terms

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

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

Fig. 3. Structural equivalence between CMS_v terms

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 [Lillibridge 1997; Ancona and Zucca 1999; Wells and Vestergaard 2000].

The notation $x_i \overset{i \in I}{\mapsto} X_i$ denotes the finite map ι such that $dom(\iota) = \{x_i \mid i \in I\}$ and 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, $cod(\iota)$ is $\{X_i \mid i \in I\}$. The finite maps $X_i \overset{i \in I}{\mapsto} E_i$ and $X_i \overset{i \in I}{\mapsto} \mathcal{T}_i$ are defined similarly.

The notions of free and bound variables, and of substitution are standard; they are defined in Figure 2.

Terms are identified up to structural equivalence, as defined in Figure 3. The equivalence rule (core-alpha) is standard α-conversion on λ-bound variables. Rule (mixin-alpha) expresses that variables bound by the input assignment of a mixin structure can be renamed if no capture occurs. In this rule, we write $\iota_1 + \iota_2$ for the unique finite map ι such that for all $x \in dom(\iota_1)$, $\iota(x) = \iota_1(x)$ and for all $x \in dom(\iota_2)$, $\iota(x) = \iota_2(x)$. This map is defined only if $\iota_1(x) = \iota_2(x)$ for all $x \in dom(\iota_1) \cap dom(\iota_2)$.

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, a 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 $X \in dom(o) \setminus cod(\iota)$.

For example, 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

$$\begin{aligned} \iota &= [x \mapsto X;\ y \mapsto Y;\ z \mapsto Z] \\ o &= [Y \mapsto x + 2;\ Z \mapsto y + 1]. \end{aligned}$$

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 inputs and outputs, 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.

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

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.

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

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

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

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 $Y \xrightarrow{\chi_1} X \xrightarrow{\chi_2} Z$ in the original graph become edges $Y \xrightarrow{\min(\chi_1,\chi_2)} Z$ in the result graph.

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

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. Remarkably, all other graph operations preserve safety.

LEMMA 3.1. *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 given 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 Ancona and Zucca [2002], 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, 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

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

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

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

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

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

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

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

Fig. 4. Typing rules for CMS_v

signature and in the output signature of the structure are identical, then removes it from the input signature. (The notation $\mathcal{I}_{\setminus X}$ denotes the finite map obtained from $\mathcal{I}$ by removing the binding for X.) 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.

Continuing the example at the end of section 3.1, the mixin $\langle \iota; o\rangle$, where $\iota = [x \mapsto X;\ y \mapsto Y;\ z \mapsto Z]$ and $o = [Y \mapsto x + 2;\ Z \mapsto y + 1]$, has type $\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$, where

$$\begin{aligned} \mathcal{I} &= [X \mapsto \mathtt{int}; Y \mapsto \mathtt{int}; Z \mapsto \mathtt{int}] \\ \mathcal{O} &= [Y \mapsto \mathtt{int}; Z \mapsto \mathtt{int}] \\ \mathcal{D} &= X \xrightarrow{0} Y \xrightarrow{0} Z. \end{aligned}$$

For simplicity, the rules (sum), (freeze) and (close) require strict syntactic equality of types. Although we will not do it here, it is possible to introduce a notion of subtyping [Ancona and Zucca 2002] 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 $\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$ is well-formed if $\mathcal{D}$ as well as the graphs appearing in $\mathcal{I}$ and $\mathcal{O}$ are safe, and moreover *Sources*($\mathcal{D}$) and *Sinks*($\mathcal{D}$), the set of nodes possessing at least one outgoing (respectively, incoming) edge, are

$$\vdash \tau \text{ (core)} \qquad \frac{\begin{array}{c} \mathit{Sources}(\mathcal{D}) \subset \mathit{dom}(\mathcal{I}) \quad \mathit{Sinks}(\mathcal{D}) \subset \mathit{dom}(\mathcal{O}) \\ \vdash \mathcal{I}(X) \text{ for all } X \in \mathit{dom}(\mathcal{I}) \quad \vdash \mathcal{O}(X) \text{ for all } X \in \mathit{dom}(\mathcal{O}) \quad \vdash \mathcal{D} \end{array}}{\vdash \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}} \text{ (mixin)}$$

Fig. 5. Well-formed CMS_v types

included in $dom(\mathcal{I})$ (respectively, $dom(\mathcal{O})$). Our type system satisfies the following well-formedness property.

LEMMA 3.2. *If* $\Gamma \vdash E : \mathcal{T}$ *is derivable, and* $\vdash \Gamma(x)$ *for all* $x \in dom(\Gamma)$*, then* $\vdash \mathcal{T}$*.*

PROOF. The proof is a simple induction on the proof tree, relying on the condition that all the dependency graphs appearing in the environment and in input signatures are safe, on lemma A.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 Overview

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 direct 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

Values

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

Evaluation contexts

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

Parallel substitution by $\rho = \ldots \ x_i \leftarrow M_i \ \ldots$

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

Reduction rules

$$\begin{array}{rcll} (\lambda x.M) \ v & \rightarrow & M\{x \leftarrow v\} & \text{(beta)} \\ \textbf{let} \ x = v \ \textbf{in} \ M & \rightarrow & M\{x \leftarrow v\} & \text{(bind)} \\ \langle X_1 = v_1 \ldots X_n = v_n \rangle.X_i & \rightarrow & v_i & \text{(select)} \\ \textbf{let rec} \ x_1 = v_1 \ldots x_n = v_n \ \textbf{in} \ M & \rightarrow & M\{x_1 \leftarrow M_1 \ldots x_n \leftarrow M_n\} & \text{(mutrec)} \end{array}$$

where $M_j = \textbf{let rec} \ x_1 = v_1 \ldots x_n = v_n \ \textbf{in} \ v_j$ for $j = 1, \ldots, n$.

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

Fig. 6. Dynamic semantics of λ_B

λ-calculi. Fortunately, Boudol [2003] 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 introduced by Boudol [2003]. Its syntax is as follows:

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

Compared with Boudol's calculus, ours lacks references and extensible records, but features mutual recursion. The dynamic semantics of this calculus is given by Boudol's reduction rules [Boudol 2003]. Although they implement a call-by-value strategy, these rules are able to evaluate correctly recursive definitions involving

$$
\begin{array}{l}
[\![(E : \mathcal{T}').X : \mathcal{T}]\!] = [\![E : \mathcal{T}']\!].X \\
[\![\langle\iota; o\rangle : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \langle X = \vec{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X)).\ [\![o(X) : \mathcal{O}(X)]\!] \mid X \in dom(\mathcal{O})\rangle \\
[\![(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}\}]\!] = \\
\quad \textbf{let } e_1 = [\![E_1 : \{\mathcal{I}_1; \mathcal{O}_1; \mathcal{D}_1\}]\!] \textbf{ in let } e_2 = [\![E_2 : \{\mathcal{I}_2; \mathcal{O}_2; \mathcal{D}_2\}]\!] \textbf{ 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 \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) \setminus X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in } \langle Y = e.Y \mid Y \in dom(\mathcal{O})\rangle \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})[X \leftarrow Y] : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \langle Z\{X \leftarrow Y\} = \vec{\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 \\
[\![(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\})\,!\,X : \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \langle Z = e.Z \mid Z \in dom(\mathcal{O}),\ X \notin \mathcal{D}'^{-1}(Z); \\
\quad\ \ Y = \vec{\lambda}\ \overline{\mathcal{D}^{-1}(Y)}.\ \textbf{let rec } \overline{X} = e.X\ \overline{\mathcal{D}'^{-1}(X)} \textbf{ in } e.Y\ \overline{\mathcal{D}'^{-1}(Y)} \mid X \in \mathcal{D}'^{-1}(Y)\rangle \\
[\![\texttt{close}(E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}) : \{\emptyset; \mathcal{O}; \emptyset\}]\!] = \\
\quad \textbf{let } e = [\![E : \{\mathcal{I}'; \mathcal{O}'; \mathcal{D}'\}]\!] \textbf{ in} \\
\quad \textbf{let rec } \overline{X^1_1} = e.X^1_1\ \overline{\mathcal{D}'^{-1}(X^1_1)} \textbf{ and } \ldots \textbf{ and } \overline{X^1_{n_1}} = e.X^1_{n_1}\ \overline{\mathcal{D}'^{-1}(X^1_{n_1})} \textbf{ in} \\
\quad \ldots \\
\quad \textbf{let rec } \overline{X^p_1} = e.X^p_1\ \overline{\mathcal{D}'^{-1}(X^p_1)} \textbf{ and } \ldots \textbf{ and } \overline{X^p_{n_p}} = e.X^p_{n_p}\ \overline{\mathcal{D}'^{-1}(X^p_{n_p})} \textbf{ 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}'
\end{array}
$$

Fig. 7. The translation scheme from CMS_v to λ_B

function applications, such as:

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

The dynamic semantics of the calculus is defined in Figure 6. The only difference from standard call-by-value evaluation is that variables are considered values. Thus, applications such as $(\lambda yz.(zy))\ x$ are redexes, and recursive definitions such as the one above can be reduced. Notice that the (mutrec) rule crucially relies on parallel capture-avoiding substitution, also defined in Figure 6.

4.3 The translation

The translation scheme for our language is defined in Figure 7. The translation, written $[\![E : \mathcal{T}]\!]$ is type-directed and operates on terms E annotated by their types $\mathcal{T}$. 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 with $o(X)$ 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 $\vec{\lambda}(x_1, \ldots, x_n).M$ 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 $\vec{\lambda}\iota^{-1}(\mathcal{D}^{-1}(X)).[\![o(X) : \mathcal{O}(X)]\!]$.

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 $\{Y_1 \ldots Y_n, X_1 \ldots X_p\} \setminus \{X\}$. 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 `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 $(\{X^1_1 \ldots X^1_{n_1}\}, \ldots, \{X^p_1 \ldots X^p_{n_p}\})$ of the exported names where each cluster $\{X^i_1 \ldots X^i_{n_i}\}$ 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.

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

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

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

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

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

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

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

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

Fig. 8. Typing rules for λ_B

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** N. In λ_B, these definitions can either evaluate to a fixpoint (for instance, $M = \lambda x.\lambda y.y$), or get stuck (for instance, $M = \lambda x.x(\lambda y.y)$). 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 [2003] gave such a type system, using annotated function types $\tau_1 \xrightarrow{0} \tau_2$ and $\tau_1 \xrightarrow{1} \tau_2$ to distinguish functions that respectively need or do not need the value of their argument immediately after application. However, Boudol's type system does not keep track of dependencies in curried functions with sufficient precision for our purposes. Hence we now define a refinement of Boudol's type system, where the annotations 0 and 1 on function types are generalized into natural integers.

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

λ_B types:	$\tau ::= \texttt{int} \mid \texttt{bool}$	base types
	$\mid \tau_1 \xrightarrow{d} \tau_2$	annotated function types

$$| \langle \ldots \; X_i : \tau_i \; \ldots \rangle \quad \text{record types}$$

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 $\mathtt{int} \xrightarrow{2} \ldots$ 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. 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 may be needed to evaluate M; and $\gamma(x) = n + 1$ means that the value of x is definitely not needed when apply M to n or fewer function applications, for instance if 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:

$$d \,@\, 0 = 0 \qquad d \,@\, \infty = \infty \qquad d \,@\, (n + 1) = d.$$

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 to how many arguments 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 $(d \,@\, \gamma_2)$ 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 premises $d_{ik} \leq d_{ij} \,@\, d_{jk}$, which we nickname the “triangular inequalities”. Finally, the dependencies of the whole **let rec** are obtained by combining those of its body

$\mathcal{D}^{-1}(X) = (X_1, \ldots, X_n)$ is the list of the predecessors of X in $\mathcal{D}$, ordered lexicographically.
$\mathcal{D}(X,Y) = \min\{\chi \mid X \xrightarrow{\chi} Y \in \mathcal{D}\}$ (with the convention that $\mathcal{D}(X,Y) = \infty$ if D contains no edges from X to Y)
$FCT_{\mathcal{D}}(X,\mathcal{I}) = (\mathcal{T}_1^{\chi_1}, \ldots, \mathcal{T}_n^{\chi_n})$, for $Sources(\mathcal{D}) \subset dom(\mathcal{I})$, where $\mathcal{D}^{-1}(X) = (X_1, \ldots, X_n)$ and for all $i \in \{1 \ldots n\}$, $\mathcal{I}(X_i) = \mathcal{T}_i$ and $\mathcal{D}(X_i, X) = \chi_i$.
$Sources(\mathcal{D}) = \{X \mid X \xrightarrow{\chi} Y \in \mathcal{D}, X, Y \in Names, \chi \in Vals\}$
$Sinks(\mathcal{D}) = \{Y \mid X \xrightarrow{\chi} Y \in \mathcal{D}, X, Y \in Names, \chi \in Vals\}$

Fig. 9. Operations on graphs

M with those arising from the uses of the x_i in M, either direct ($d_i \,@\, \gamma_i$) or one-step indirect ($d_i \,@\, d_{ij} \,@\, \gamma_j$). Longer indirect dependencies such as $d_i \,@\, d_{ij} \,@\, d_{jk} \,@\, \gamma_k$ need not be taken into account because of the triangular inequalities.

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 5.1. *(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 theorem follows from the following lemmas, which are proved in appendix B. The first three lemmas are substitution lemmas for general one-variable substitution, substitution of one variable by another, and parallel substitution. They play a crucial role for proving subject reduction for the typing rules (app), (appvar) and (rec) respectively.

Lemma 5.2. *(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 $\Gamma \vdash M_1\{x \leftarrow M_2\} : \tau \,/\, \gamma_1 \wedge d \,@\, \gamma_2$.*

Lemma 5.3. *(Substitution by a variable.) If $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \,/\, \gamma[x \mapsto d]$ and $\Gamma(y) = \tau'$, then $\Gamma \vdash M\{x \leftarrow y\} : \tau \,/\, \gamma \wedge (y \mapsto d)$.*

Lemma 5.4. *(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 $\Gamma \vdash M\{\ldots\ x_i \leftarrow M_i\ \ldots\} : \tau \,/\, \gamma_M \wedge \bigwedge_i d_i \,@\, \gamma_i$.*

We then show the standard properties of subject reduction (reduction preserves typing) and progress (well-typed terms are not stuck).

Lemma 5.5. *(Subject reduction.) If $\Gamma \vdash M : \tau \,/\, \gamma$ and $M \to M'$, then $\Gamma \vdash M' : \tau \,/\, \gamma$.*

Lemma 5.6. *(Progress.) If $\Gamma \vdash M : \tau \,/\, \gamma$ and $\gamma \geq 1$, then either M is a value, or there exists M' such that $M \to M'$.*

The soundness of λ_B follows from lemmas 5.5 and 5.6. □

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 translates to a well-typed λ_B expression. The soundness of λ_B then ensures that the translation evaluates correctly.

$$
\begin{array}{rcl}
[\![\tau_1 \to \tau_2]\!] & = & \tau_1 \xrightarrow{0} \tau_2 \\
[\![\texttt{int}]\!] & = & \texttt{int} \\
[\![\texttt{bool}]\!] & = & \texttt{bool} \\
[\![\{\mathcal{I};\mathcal{O};\mathcal{D}\}]\!] & = & \langle X : [\![\mathcal{O}(X)]\!]_{X,\mathcal{D},\mathcal{I}} \mid X \in dom(\mathcal{O})\rangle \text{ if } \vdash \{\mathcal{I};\mathcal{O};\mathcal{D}\} \\
[\![\mathcal{T}]\!]_{X,\mathcal{D},\mathcal{I}} & = & [\![\mathcal{T}_1]\!] \xrightarrow{\chi_1+(n-1)} [\![\mathcal{T}_2]\!] \xrightarrow{\chi_2+(n-2)} \ldots [\![\mathcal{T}_n]\!] \xrightarrow{\chi_n} [\![\mathcal{T}]\!] \\
 & & \text{where } (\mathcal{T}_1^{\chi_1}, \ldots, \mathcal{T}_n^{\chi_n}) = FCT_{\mathcal{D}}(X,\mathcal{I})
\end{array}
$$

Fig. 10. Translation of CMS_v types into λ_B types

Core terms:	$\overline{C} ::= x^{\mathcal{T}} \mid cst^{\mathcal{T}}$	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:	$\bar{o} ::= X_i \overset{i \in I}{\mapsto} \overline{E}_i$	

Fig. 11. Syntax of type-annotated CMS_v terms

To state the soundness of the translation, we need to set up a translation from source types to λ_B types. We start by defining useful operations on 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, $Sources(\mathcal{D})$ and $Sinks(\mathcal{D})$ 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, $\min(\chi_1, \chi_2) = \chi_1 \wedge \chi_2 = \chi_1 \mathbin{@} \chi_2$.

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 annotated terms in the following.

We define $IsRec(E)$ as 1 if E is an abstraction $\lambda x.C$, and 0 otherwise, and extend this definition to annotated expressions.

Theorem 5.7. *(Soundness of the translation.) If* $\Gamma \vdash E : \mathcal{T}$*, then* $[\![\Gamma]\!] \vdash [\![\overline{E}]\!] : [\![\mathcal{T}]\!] \,/\, d^o(\Gamma) + IsRec(E)$.

See appendix C for the full proof. Notice that this result holds for non-empty contexts Γ; in conjunction with the compositional nature of the translation, this ensures that our compilation scheme is applicable (and sound) not only to closed programs, but also to terms with free variables as can arise during separate compilation.

6. RELATED WORK

6.1 Mixin-based inheritance and object-oriented traits

The notion of mixin originates in the object-oriented language Flavors [Moon 1986], and was further investigated both as a linguistic device addressing many of the shortcomings of inheritance [Flatt et al. 1998; Findler and Flatt 1998] and as a semantic foundation for inheritance [Cook 1989]. An issue with mixin classes that is generally not addressed is the treatment of instance fields and their initialization. Mixin classes where instance fields can be initialized by arbitrary expressions raise exactly the same problems of detecting cyclic dependencies that we have addressed in this paper in the context of call-by-value mixin modules. Initialization can also be performed by an initialization method named `init` or some other conventional name, but this breaks data encapsulation.

The notion of traits [Black et al. 2003] shares several key features with mixin modules. Traits are collections of named methods that can be combined together and with regular class definitions using various operators such as sum, overriding, aliasing and exclusion. Traits contain only methods but not instance fields; therefore, initialization of instance fields is again not addressed.

6.2 Language designs with mixin modules

Bracha [Bracha 1992] formulated the concept of mixin-based inheritance (sum) independently of an object-oriented setting. His mixins do not address the initialization issue. Duggan and Sourelis [1996] transposed Bracha's mixin concept to the ML module system. Their mixin module system supports extensible functions and datatypes: a function defined by cases 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 recursion problem 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.

Flatt and Felleisen [1998] introduce the closely related concept of *units*, which adapt Bracha's ideas to Scheme and ML. A first difference with our proposal is that units do not feature late binding. Moreover, the initialization problem is handled differently. Their implementation of units for Scheme allows arbitrary computations within the definitions of unit components, and evaluates these computations like Scheme's `letrec` construct. Thus, ill-founded recursions are not prevented statically. The formalization of units in [Flatt and Felleisen 1998, section 4] restricts definitions to syntactic values, but includes an initialization expression in each unit. This initialization expression can perform arbitrary computations and refer to the variables bound by the definitions, but is evaluated for its side-effects only. As in Duggan and Sourelis' system, this approach prevents the creation of ill-founded recursive definitions, but is less flexible than our approach.

6.3 Mixin calculi

Ancona and Zucca [1998; 1999; 2002] 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 the `m`-calculus [Wells and Vestergaard 2000], which is very similar to *CMS* in many aspects, 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 and Zucca nor Wells and Vestergaard attempt to statically control recursive definitions, performing on-demand unwinding instead. Still, some care is required when unwinding definitions inside a structure, because of confluence problems [Ariola and Blom 2002].

6.4 Recursive modules in ML

Crary et al. [1999], Dreyer et al. [2001], and Russo [2001] extend the Standard ML module system with mutually recursive structures via a `structure rec` 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 separate compilation can be recovered 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 run-time tests to catch them during evaluation. 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.

6.5 Type systems for well-founded recursion

The type system for λ_B presented in section 5 is a refinement of the type system introduced by Boudol [2003]. Dreyer [2004] and Dreyer et al. [2003] propose a different type system to guarantee safe call-by-value evaluation of generalized recursive definitions of the form **let rec** $x = M\ x$ **in** N. Their system can be viewed as an effect system that tracks the (pro forma) effect of using the value of a recursively-bound variable. The typing rules ensure that no such use can occur before the recursive definition has been fully evaluated. This type system appears expressive enough to show that the terms produced by our compilation scheme do not get stuck on an illegal recursive definition. Moreover, its type soundness proof appears simpler than that of our type system. A drawback of Dreyer's system for our purpose is that it requires "boxing" and "unboxing" annotations in terms and in type expressions. It is not immediately obvious how to extend the compilation scheme given in section 4 to insert the correct annotations.

6.6 Connections with object-oriented type systems

Bono et al. [1999] use a notion of dependency graph in the context of a type system for extensible and incomplete objects. However, they do not distinguish between "0" and "1" dependencies, since the fact that objects contain only methods but no immediate computations precludes immediate dependencies between methods.

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.

Our λ_B target calculus can be compiled efficiently down to machine code, using the "in-place updating" trick outlined in [Cousineau et al. 1987] and formalized in [Hirschowitz et al. 2003; Hirschowitz 2003] to implement the non-standard `let rec` construct.

In this paper, the dynamic semantics of CMS_v is given by translation. A direct reduction semantics is desirable to allow finer reasoning on the evaluation of mixins. More recent work [Hirschowitz et al. 2004; Hirschowitz 2003] develops a call-by-value reduction semantics for a calculus of mixins called MM, closely related to CMS_v.

The translation semantics of CMS_v raises another issue that is better addressed in the reduction semantics of MM: programmer control of evaluation order. In CMS_v, when a mixin is closed, its definitions are evaluated in an order that is only partially determined by a topological sort of its dependency graph. Moreover, the freeze operator duplicates the definition of the frozen component into the components that use it, resulting in multiple evaluations of the frozen component later. These two features are problematic when the core language is imperative. In MM, frozen components are never duplicated, but turned into local (nameless) definitions instead; and the evaluation order of components is unambiguously determined by a combination of the initial ordering of definitions in structures, and programmer-supplied "fake dependency" annotations on definitions.

The price to pay for this better control of evaluation order is that MM does not lend itself to a type-directed compilation scheme like the one presented in this paper. Since local definitions do not appear in mixin signatures, it is not possible to determine when and where they should be evaluated based on the signatures of the mixins involved, like the compilation scheme presented in this paper does. Indeed, the only known implementation scheme for MM is interpretative in nature and relies on run-time interpretation of dependency graphs. The overhead of this interpretation is acceptable if mixins are second-class (like ML modules), but if mixins are first-class values, the compilation scheme for CMS_v presented here is much more efficient.

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. A more ambitious approach is to infer dependency graphs entirely, by generating constraints between formal variables ranging over dependency graphs, and solving these constraints incrementally.

The next step towards mixins 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 body of type-theoretic work already done for ML modules [Harper and Lillibridge 1994; Leroy 1994] and recursive modules [Crary et al. 1999; Dreyer et al. 2001].

ACKNOWLEDGMENTS

We thank Elena Zucca and Davide Ancona for discussions, Vincent Simonet for his technical advice on the typing rules for λ_B, and the anonymous reviewers for their helpful suggestions in improving the presentation of this paper.

REFERENCES

Abadi, M. and Cardelli, L. 1996. *A theory of objects.* Springer-Verlag, Berlin.

Ancona, D. 1998. Modular formal frameworks for module systems. Ph.D. thesis, Universita di Pisa.

Ancona, D. and Zucca, E. 1999. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming, International Conference PPDP'99.* Lecture Notes in Computer Science, vol. 1702. Springer-Verlag, Berlin, 62–79.

Ancona, D. and Zucca, E. 2002. A calculus of module systems. *Journal of Functional Programming 12,* 2, 91–132.

Ariola, Z. M. and Blom, S. 2002. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic 117,* 1–3, 95–178.

Black, A. P., Schärli, N., and Ducasse, S. 2003. Applying traits to the Smalltalk collection classes. In *Conference on Object-Oriented Programming Systems, Languages and Applications 2003.* ACM Press, New York, NY, 47–64.

Bono, V., Bugliesi, M., Dezani-Ciancaglini, M., and Liquori, L. 1999. Subtyping for extensible, incomplete objects. *Fundamenta Informaticae 38,* 4, 325–364.

Boudol, G. 2003. The recursive record semantics of objects revisited. *Journal of Functional Programming 14,* 3, 263–315.

Bracha, G. 1992. The programming language Jigsaw: mixins, modularity and multiple inheritance. Ph.D. thesis, University of Utah.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages and Applications 1990.* SIGPLAN Notices, vol. 25(10). ACM Press, New York, NY, 303–311.

Cardelli, L. 1997. Program fragments, linking, and modularization. In *24th symposium Principles of Programming Languages.* ACM Press, New York, NY, 266–277.

Cook, W. R. 1989. A denotational semantics of inheritance. Ph.D. thesis, Department of Computer Science, Brown University.

Cousineau, G., Curien, P.-L., and Mauny, M. 1987. The categorical abstract machine. *Science of Computer Programming 8,* 2, 173–202.

Crary, K., Harper, R., and Puri, S. 1999. What is a recursive module? In *Programming Language Design and Implementation 1999.* ACM Press, New York, NY, 50–63.

Dreyer, D. 2004. A type system for well-founded recursion. In *31st symposium Principles of Programming Languages.* ACM Press, New York, NY, 293–305.

Dreyer, D., Harper, R., and Crary, K. 2003. A type system for well-founded recursion. Tech. Rep. CMU-CS-03-163, Carnegie Mellon University.

Dreyer, D. R., Crary, K., and Harper, R. 2001. Toward a practical type theory for recursive modules. Tech. Rep. CMU-CS-01-112, Carnegie Mellon University.

Duggan, D. and Sourelis, C. 1996. Mixin modules. In *International Conference on Functional Programming 1996.* ACM Press, New York, NY, 262–273.

Findler, R. B. and Flatt, M. 1998. Modular object-oriented programming with units and mixins. In *International Conference on Functional Programming 1998.* ACM Press, New York, NY, 94–104.

Flatt, M. and Felleisen, M. 1998. Units: cool modules for HOT languages. In *Programming Language Design and Implementation 1998.* ACM Press, New York, NY, 236–248.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1998. Classes and mixins. In *25th symposium Principles of Programming Languages.* ACM Press, New York, NY, 171–183.

Harper, R. and Lillibridge, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages.* ACM Press, New York, NY, 123–137.

Hirschowitz, T. 2003. Modules mixins, modules et récursion étendue en appel par valeur. Ph.D. thesis, University Paris 7.

Hirschowitz, T., Leroy, X., and Wells, J. B. 2003. Compilation of extended recursion in call-by-value functional languages. In *International Conference on Principles and Practice of Declarative Programming 2003.* ACM Press, New York, NY, 160–171.

Hirschowitz, T., Leroy, X., and Wells, J. B. 2004. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004.* Lecture Notes in Computer Science, vol. 2986. Springer-Verlag, Berlin, 64–78.

Leroy, X. 1994. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages.* ACM Press, New York, NY, 109–122.

Lillibridge, M. 1997. Translucent sums : a foundation for higher-order module systems. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Moon, D. A. 1986. Object-oriented programming with Flavors. In *Conference on Object-Oriented Programming Systems, Languages and Applications 1986.* SIGPLAN Notices, vol. 21(11). ACM Press, New York, NY, 1–8.

Russo, C. 2001. Recursive structures for Standard ML. In *International Conference on Functional Programming 2001.* ACM Press, New York, NY, 50–61.

Wells, J. B. and Vestergaard, R. 2000. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000.* Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, 412–428.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/toplas/2005-27-5/p1-Hirschowitz.pdf`.

Received May 2002; revised November 2003; accepted May 2004.

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Mixin modules in a call-by-value setting

TOM HIRSCHOWITZ
ENS Lyon
and
XAVIER LEROY
INRIA Rocquencourt

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 5, September 2005, Pages 1–24.

A. SOUNDNESS OF GRAPH OPERATIONS

In the following, we write $\mathit{fst}(P)$ and $\mathit{last}(P)$ for the first (respectively, last) node of a path p. We write $[X]$ for the zero-length path consisting of node X. If $\mathit{fst}(P) = Y$, we write $(X \xrightarrow{\chi} Y) :: P$ for the path obtained by prepending the edge $X \xrightarrow{\chi} Y$ to the path p. The valuation $\nu(P)$ of a path P is defined inductively by $\nu([X]) = 1$ and $\nu((X \xrightarrow{\chi} Y) :: P) = \min(\chi, \nu(P))$. Thus, a graph $\mathcal{D}$ is safe if and only if all paths p of $\mathcal{D}$ such that $\mathit{fst}(P) = \mathit{last}(P)$ are such that $\nu(P) = 1$.

LEMMA A.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.*

PROOF. For each operation, we show that for all path in the result graph, there exists a corresponding path with the same valuation in $\mathcal{D}$.

Renaming: 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 the length of P, we find a path with same valuation in $\mathcal{D}$, such that $\mathit{fst}(P) = A\{Y \leftarrow X\}$ and $\mathit{last}(P) = B\{Y \leftarrow X\}$.

Consider first the base case $P = [Z]$ for some name Z mentioned in $\mathcal{D}'$. 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 $\mathcal{D}$. Hence, 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.

Now, assume the result for P' and consider $P = (A \xrightarrow{\chi} B) :: P'$, with $\mathit{fst}(P') = B$. 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'. By definition of $\mathcal{D}'$ the edge

Degrees

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

Minimum

$$\begin{array}{rcl} d \wedge \infty & = & d \\ \infty \wedge d & = & d \\ m \wedge n & = & \min(m,n) \end{array}$$

Composition

$$\begin{array}{rcl} d \mathbin{@} \infty & = & \infty \\ d \mathbin{@} 0 & = & 0 \\ d \mathbin{@} n+1 & = & d \end{array}$$

Plus

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

Minus

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

Fig. 12. Summary of degree operations

$A\{Y \leftarrow X\} \xrightarrow{\chi} B\{Y \leftarrow X\}$ is in $\mathcal{D}$. Therefore, the path $(A\{Y \leftarrow X\} \xrightarrow{\chi} B\{Y \leftarrow X\}) :: P''$ is in $\mathcal{D}$ as well. It has the expected first and last nodes, and its valuation is $\min(\chi, \chi') = \nu(P)$.

It follows that every cycle in $\mathcal{D}'$ corresponds to a cycle in $\mathcal{D}$ with the same valuation. Since $\mathcal{D}$ is safe, $\mathcal{D}'$ is safe as well.

Deletion: The result is straightforward, since all edges of the resulting graph $\mathcal{D}'$ are already present in $\mathcal{D}$.

Freezing: 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 the length of P, we construct a path from A to B in $\mathcal{D}$ with the same valuation.

For the base case $P = [A]$, we have $A = B$. Since the freezing operation does not introduce new names, all names appearing in $\mathcal{D}'$ are already in $\mathcal{D}$; therefore, P is also a path of $\mathcal{D}$, obviously with valuation 1.

Consider now $P = (A \xrightarrow{\chi} C) :: P'$, with $\mathit{fst}(P') = C$ and $\mathit{last}(P') = B$. By induction hypothesis, there is a path P'' in $\mathcal{D}$ from C to B such that $\nu(P'') = \nu(P')$. We now argue by cases on the edge $A \xrightarrow{\chi} C$: by definition of the freeze operation, it can either be in $\mathcal{D}$ or in $\mathcal{D}_{around}$. If the edge $A \xrightarrow{\chi} C$ comes from $\mathcal{D}$, the path $A \xrightarrow{\chi} C :: P''$ is then clearly a path of $\mathcal{D}$, with the expected valuation and endpoints. If the edge $A \xrightarrow{\chi} C$ comes 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 $\chi = \min(\chi_1, \chi_2)$. Hence, 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 $\min(\min(\chi_1, \chi_2), \nu(P'')) = \min(\chi, \nu(P')) = \nu(P)$. $\square$

B. SOUNDNESS OF THE TARGET LANGUAGE

To simplify the proofs, we prove the soundness on a subset $\lambda_{\overline{B}}$ of λ_B that excludes constants, record construction and access, and the **let** binding. It is entirely straightforward to extend the proofs to the omitted constructs.

B.1 Properties of degrees

We start the proof with a number of algebraic lemmas on degrees and degree operations. Figure 12 re-states the definitions of the operations on degrees. The following lemmas should be read as universally quantified over the degrees d, d', d_1, d_2, d_3. We adopt the convention that @ has highest precedence, followed by $\wedge$, and then

$+$ and $-$.

LEMMA B.1.

(*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. If $d_2 = \infty$ we obtain $\infty \leq \infty$. Otherwise, the claim reduces to $d_1 + 1 \leq d_1 + 1$.
(2) If $d_3 = 0$, we obtain 0 on both sides of the equality. If $d_3 = \infty$, both sides are equal to ∞. Otherwise we get $d_1 \wedge d_2$ on both sides.
(3) If $d_2 = 0$, both sides are equal to 0. If $d_2 = \infty$, then $d_2 \wedge d_3 = d_3$ and $d_1 @ d_2 = \infty$, so both sides are equal to $d_1 @ d_3$. Otherwise, we argue 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 sides. 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 sides are equal to 0. If $d_3 = \infty$, we obtain ∞ on both sides. Otherwise, both sides are equal to $d_1 @ d_2$.
(5) Both sides reduce to ∞ if $d' = \infty$, to 0 if $d' = 0$, and to $d - 1$ otherwise.
(6) By definition of $+$.
(7) By definition of $+$ and $-$.
(8) By definition of @.
(9) By definition of $+$.
(10) Since $d + 1$ is strictly positive, d' cannot be 0. Thus, $d' = d' - 1 + 1$ by property 7, and the result follows by applying property 9 to $d + 1 \leq d' - 1$.
(11) If $d_3 = \infty$ or $d_3 = 0$, both sides reduce to d_3. Otherwise, write $d_3 = n + 1$. Then, $d_1 @ d_3 = d_1$ and $d_1 @ d_2 @ d_3 = d_1 @ d_2$, hence it simply remains to prove that $d_1 \leq d_1 @ d_2$. Since $d_2 \geq 1$, we have only two cases: either $d_2 = \infty$, in which case $d_1 @ d_2 = \infty$ which cannot be less than d_1; or $d_2 = m + 1$, in which case $d_1 @ d_2 = d_1$, and the result holds.

This completes the proof. $\square$

LEMMA B.2. *If* $\gamma \leq (\gamma_1 - 1) \wedge d @ \gamma_2$, *then there exists* γ'_1 *and* γ'_2 *such that* $\gamma = (\gamma'_1 - 1) \wedge d @ \gamma'_2$ *and* $\gamma'_1 \leq \gamma_1$ *and* $\gamma'_2 \leq \gamma_2$.

Proof. We define γ_1' and γ_2' pointwise. Consider a variable x. Let $d' = \gamma(x)$, $d_1 = \gamma_1(x)$, $d_2 = \gamma_2(x)$. We construct d_1' and d_2' such that $d' = (d_1'-1) \wedge d @ d_2'$ and $d_1' \leq d_1$ and $d_2' \leq d_2$. If $d' = 0$, then we can take $d_1' = d_2' = 0$. If $d' = \infty$, then we can take $d_1' = d_1$ and $d_2' = d_2$, because only ∞ is greater than d'. Finally, if $d' = n+1$, let $d_1' = n+2$ and $d_2' = d_2$. By hypothesis we know that $d' \leq d @ d_2$. Since $d_1' - 1 = n+1 = d'$, we have $(d_1'-1) \wedge d @ d_2' = d_1' - 1 = d'$. Moreover, since $d' \leq d_1 - 1$, we have that $n+1 \leq d_1 - 1$, and therefore $(d_1' = n+2 \leq d_1$ by lemma B.1. Finally, $d_2' \leq d_2$ trivially holds. □

Lemma B.3. *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 as in the previous proof. Consider a variable y and let $d' = \gamma(y)$ and $d_1 = \gamma_1(y)$. We construct 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, we take $d_1' = d' + 1$. This definition satisfies the following properties:

—Since $d' \leq d_1 - 1$, we have $d' + 1 \leq d_1 - 1 + 1$ and $d_1 \neq 0$. By lemma B.1, it follows that $d_1 - 1 + 1 = d_1$, hence $d_1' \leq d_1$.
—From $d' \leq (d' + 1 - 1) \leq (d_1' - 1)$ and $d' \leq (d_1 - 1) \wedge (x \mapsto d)(y) \leq (x \mapsto d)(y)$, it follows that $d' \leq (d_1' - 1) \wedge ((x \mapsto d)(y))$.
—Since $d_1' - 1 = d'$, we have that $(d_1' - 1) \wedge ((x \mapsto d)(y)) \leq d'$.

Thus, d_1' satisfies the claim. □

Lemma B.4. *Let $n \in \mathbb{N}$. If*

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

then there exist $\gamma_0', \gamma_1', \ldots, \gamma_n'$ such that $\gamma_i' \leq \gamma_i$, for $i = 0, \ldots, n$ and

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

Proof. Simply take $\gamma_0' = \gamma'$ and $\gamma_i' = \gamma_i$ for $i = 1, \ldots, n$. By transitivity we have $\gamma_0' \leq \gamma_0$ and trivially $\gamma_i' \leq \gamma_i$. It is easy to check that

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

by definition of γ'. Moreover, by hypothesis, we know that

$$\bigwedge_{i,j \in \{1 \ldots n\}} d_i @ d_{ij} @ \gamma_j' \geq \gamma' \qquad \text{and} \qquad \bigwedge_{i \in \{1 \ldots n\}} d_i @ \gamma_i' \geq \gamma'.$$

Therefore,

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

and the expected equality follows. □

Lemma B.5. *If $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 @ \gamma_2$ 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 $\gamma = (\gamma_1' - 1) \wedge d_0 @ \gamma_2'$.*

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,

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

On $y \neq x$,

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

This is the expected result. □

LEMMA B.6. *If* $\gamma[x \mapsto d] = \gamma_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i @ d_{ij} @ \gamma_j\big) \wedge \big(\bigwedge_i d_i @ \gamma_i\big)$*, 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* $\gamma = \gamma'_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i @ d_{ij} @ \gamma'_j\big) \wedge \big(\bigwedge_i d_i @ \gamma'_i\big)$*, with* $d_0 = \gamma_0(x)$ *and* $d'_i = \gamma_i(x)$ *for all* i*.*

PROOF. Take $\gamma'_0 = \gamma_0[x \mapsto \gamma(x)]$ and $\gamma'_i = \gamma_i[x \mapsto \infty]$ for all i. We check that the expected properties hold as in the previous proof. □

B.2 Weakening lemmas

We now prove two "weakening" lemmas showing that the typing judgement still holds if the degree environment γ is replaced by another environment $\gamma' \leq \gamma$, or if the degree $\gamma(x)$ of an unused variable x is changed.

LEMMA B.7. *(Degree restriction.) If* $\gamma' \leq \gamma$ *and* $\Gamma \vdash M : \tau \ / \ \gamma$*, then* $\Gamma \vdash M : \tau \ / \ \gamma'$*.*

PROOF. We reason by induction on the typing derivation of M, and by case on the last typing rule used. (Refer to Figure 8 for the typing rules of λ_B.)

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

Rule(abstr), $M = \lambda x.M_1$. Given the typing rules, we have a derivation of $\Gamma + \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma - 1)[x \mapsto d]$ with $\tau = \tau_1 \xrightarrow{d} \tau_2$. Notice that $(\gamma' - 1)[x \mapsto d] \leq (\gamma - 1)[x \mapsto d]$. Therefore, by induction hypothesis, we have a derivation of $\Gamma + \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma' - 1)[x \mapsto d]$. The expected result follows by another application of the rule (abstr).

Rule (app), $M = M_1 \ M_2$. By typing hypothesis, 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 B.2, we construct γ'_1 and γ'_2, such that $\gamma'_1 \leq \gamma_1$, $\gamma'_2 \leq \gamma_2$ and $\gamma' = (\gamma'_1 - 1) \wedge d @ \gamma'_2$. Applying the induction hypothesis twice, we obtain 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 expected result.

Rule (appvar), $M = M_1 \ x$. We have a derivation for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \ / \ \gamma_1$ with $\Gamma(x) = \tau'$ and $\gamma = (\gamma_1 - 1) \wedge d$. Hence, $\gamma' \leq (\gamma_1 - 1) \wedge (x \mapsto d)$. Applying lemma

B.3, we obtain γ'_1 such that $\gamma'_1 \leq \gamma_1$ and $\gamma' = (\gamma'_1 - 1) \wedge (x \mapsto d)$. We can apply rule (appvar) again to derive the expected judgment.

Rule (rec), $M = \textbf{let rec} \ \dots \ x_i = M_i \ \dots \ \textbf{in} \ N$. By typing hypothesis, we have

$$\begin{gathered}
\Gamma + \{\dots \ x_j : \tau_j \ \dots\} \vdash N : \tau \ / \ \gamma_0[\dots \ x_j \mapsto d_j \ \dots] \\
\Gamma + \{\dots \ x_j : \tau_j \ \dots\} \vdash M_i : \tau_i \ / \ \gamma_i[\dots \ x_j \mapsto d_{ij} \ \dots] \\
\text{for all } i, j, \ d_{ij} \geq 1 \\
\text{for all } i, j, k, \ d_{ik} \leq d_{ij} \mathbin{@} d_{jk} \\
\gamma = \gamma_0 \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big).
\end{gathered}$$

Using lemma B.4, we take $\gamma'_N = \gamma'$ 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 obtain a derivation of $\Gamma + \{\dots \ x_j : \tau_j \ \dots\} \vdash N : \tau \ / \ \gamma'_N[\dots \ x_j \mapsto d_j \ \dots]$. Hence we can derive $\Gamma \vdash M : \tau \ / \ \gamma'$. □

Lemma B.8. *(Degree weakening.) If $\Gamma \vdash M : \tau \ / \ \gamma[x \mapsto d]$ and $x \notin FV(M)$, then $\Gamma \vdash M : \tau \ / \ \gamma$.*

Proof. The proof is by induction on the typing derivation of M and by case on the last rule used.

Rule (var), $M = y$. Since $x \notin FV(M)$, we have $x \neq y$. By typing hypotheses, $\gamma(y) = 0$ and $\Gamma(y) = \tau$. It follows that $\Gamma \vdash M : \tau \ / \ \gamma$.

Rule (abstr), $M = \lambda y.M_1$, where y is fresh. The premise of the typing rule holds: $\Gamma + \{y \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma[x \mapsto d] - 1)[y \mapsto d_0]$ and $\tau = \tau_1 \xrightarrow{d_0} \tau_2$. Obviously, $(\gamma[x \mapsto d] - 1)[y \mapsto d_0] = (\gamma - 1)[y \mapsto d_0][x \mapsto d - 1]$. Hence, by induction hypothesis we obtain $\Gamma + \{y \mapsto \tau_1\} \vdash M_1 : \tau_2 \ / \ (\gamma - 1)[y \mapsto d_0]$ and the expected result follows by rule (abstr).

Rule (app), $M = M_1 \ M_2$. We have $\Gamma \vdash M_1 : \tau' \xrightarrow{d_0} \tau \ / \ \gamma_1$ and $\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$. Applying lemma B.5, we obtain 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 $\Gamma \vdash M_2 : \tau' \ / \ \gamma'_2$. The expected result follows by rule (app).

Rule (appvar), $M = M_1 y$, with $y \neq x$ by hypothesis $x \notin FV(M)$. We have a derivation of $\Gamma \vdash M_1 : \tau_1 \xrightarrow{d_0} \tau_2 \ / \ \gamma_1$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge (y \mapsto d_0)$. Take $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. We have $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$ and $\gamma = (\gamma'_1 - 1) \wedge (y \mapsto d_0)$. The first equality is straightforward, and the second equality follows from the facts that $\gamma(x) = \gamma(x) + 1 - 1$, and for any $z \neq x$, $((\gamma_1 - 1) \wedge (y \mapsto d_0))(z) = ((\gamma'_1 - 1) \wedge (y \mapsto d_0))(z)$. We then conclude by induction hypothesis as above.

Rule (rec), $M = \textbf{let rec} \ \dots \ x_i = M_i \ \dots \ \textbf{in} \ N$. We have

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

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 $\gamma[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$. Lemma B.6 shows the existence of γ'_N and 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 $\gamma = \gamma'_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big)$, with $d_N = \gamma_N(x)$ and for all i, $d'_i = \gamma'_i(x)$. Applying the 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

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

The result follows by rule (rec). $\square$

LEMMA B.9. *(Type weakening.) If* $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ /\ \gamma$ *and* $x \notin FV(M)$, *then* $\Gamma \vdash M : \tau \ /\ \gamma$.

PROOF. Straightforward by induction on the typing derivation. $\square$

B.3 Substitution lemmas

We now establish the traditional substitution lemma: a variable can be substituted by a term of the same type without affecting the type of the program. This lemma provides a semantic justification for our definition of @ in relation with what actually happens during the reduction of an application.

LEMMA B.10. *(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 and case analysis on the last typing rule used. We write $M = M_1\{x \leftarrow M_2\}$, $\Gamma' = \Gamma + \{x \mapsto \tau'\}$, and $\gamma_0 = \gamma_1 \wedge d \mathbin{@} \gamma_2$.

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

If $y = x$, then $M = M_2$, $d = 0$, $\tau = \tau'$ and by hypothesis $\Gamma \vdash M : \tau \ /\ \gamma_2$. By lemma B.7, it suffices to show that $\gamma_0 \leq \gamma_2$ or $\gamma_1 \wedge 0 \mathbin{@} \gamma_2 \leq \gamma_2$, which is true by lemma B.1.

If $y \neq x$, then $x \notin FV(M)$ and $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ /\ \gamma_1[x \mapsto d]$. By lemmas B.8 and B.9, $\Gamma \vdash M : \tau \ /\ \gamma_1$. It suffices to show that $\gamma_0 \leq \gamma_1$, which is trivially true.

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

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

with $\tau = \tau_1 \xrightarrow{d_0} \tau_2$ and $\gamma_3[y \mapsto d_0] = (\gamma_1[x \mapsto d] - 1)[y \mapsto d_0] = (\gamma_1 - 1)[x \mapsto (d-1); y \mapsto d_0]$. Take $M'_3 = M_3\{x \leftarrow M_2\}$. By induction hypothesis, we have

$\Gamma + \{y \mapsto \tau_1\} \vdash M_3' : \tau_2 \ / \ (\gamma_1 - 1)[y \mapsto d_0] \wedge (d-1) \mathbin{@} \gamma_2$. Since y is fresh, it does not occur in γ_2, therefore

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

Hence, rule (abstr) concludes $\Gamma \vdash \lambda y.M_3' : \tau_1 \xrightarrow{d_0} \tau_2 \ / \ \gamma_0$, which is the expected result.

Rule (app), $M_1 = M_3 \ M_4$. 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 B.5, if $d_3 = \gamma_3(x)$ and $d_4 = \gamma_4(x)$, there exists 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 $\Gamma \vdash M_4' : \tau'' \ / \ \gamma_4' \wedge d_4 \mathbin{@} \gamma_2$. Therefore, by rule (app),

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

Moreover, by lemma B.1, the degree environment is equal to

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

Rule (appvar), $M_1 = M_3 \ y$. As in the (var) case, we argue by case, according to whether y is equal to x or not.

Case $y = x$. Here, $M = M_3' \ M_2$, where $M_3' = M_3\{x \leftarrow M_2\}$. The typing hypothesis implies $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3$ (*) and $\Gamma'(y) = \Gamma'(x) = \tau' = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Take $\gamma_3' = \gamma_3[x \mapsto \gamma_1(x) + 1]$. We have $\gamma_1 = (\gamma_3' - 1)$ and $\gamma_3'[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we can write the premise (*) as follows

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

Hence, 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 obtain

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

Notice that $\gamma_0 = (\gamma_3' - 1) \wedge d \mathbin{@} \gamma_2$. Since $d = (d_3 - 1) \wedge d_0$, it follows that

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

We therefore have derived the desired judgment.

Case $y \neq x$. Here, $M = M'_3\ y$, where $M'_3 = M_3\{x \leftarrow M_2\}$. By typing hypothesis, we have $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3$ (*) and $\Gamma'(y) = \Gamma(y) = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Take $\gamma'_3 = \gamma_3[x \mapsto \gamma_1(x) + 1]$. We have $\gamma_1 = (\gamma'_3 - 1) \wedge (y \mapsto d_0)$, and $\gamma'_3[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we rewrite the premise (*) as follows:

$$\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma'_3[x \mapsto \gamma_3(x)].$$

By induction hypothesis, it follows that

$$\Gamma \vdash M'_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma'_3 \wedge d_3 \ @ \ \gamma_2$$

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

$$\Gamma \vdash M : \tau \ / \ ((\gamma'_3 \wedge d_3 \ @ \ \gamma_2) - 1) \wedge (y \mapsto d_0)$$

which yields by lemma B.1

$$\Gamma \vdash M : \tau \ / \ (\gamma'_3 - 1) \wedge (d_3 \ @ \ \gamma_2 - 1) \wedge (y \mapsto d_0).$$

Moreover,

$$\begin{aligned}
\gamma_0 &= \gamma_1 \wedge d \ @ \ \gamma_2 \\
&= (\gamma'_3 - 1) \wedge (y \mapsto d_0) \wedge d \ @ \ \gamma_2 \\
&= (\gamma'_3 - 1) \wedge (y \mapsto d_0) \wedge (d_3 - 1) \ @ \ \gamma_2 \\
&\text{(because } \gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)) \\
&= (\gamma'_3 - 1) \wedge (y \mapsto d_0) \wedge (d_3 \ @ \ \gamma_2 - 1) \qquad \text{(by lemma B.1).}
\end{aligned}$$

Thus, the expected result holds.

Rule (rec), $M =$ **let rec** $x_1 = N_1$ **and** $\ldots$ **and** $x_n = N_n$ **in** N, where the x_i are fresh. By typing hypothesis,

$$\begin{gathered}
\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \ / \ \gamma_N[\ldots\ x_j \mapsto d_j\ \ldots] \\
\text{for all } i,\ \Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N_i : \tau_i \ / \ \delta_i[\ldots\ x_j \mapsto d_{ij}\ \ldots] \\
\text{for all } i,\ j,\ d_{ij} \geq 1 \\
\text{for all } i,\ j,\ k,\ d_{ik} \leq d_{ij} \ @ \ d_{jk}
\end{gathered}$$

We write $N' = N\{x \leftarrow M_2\}$ and for all i, $N'_i = N_i\{x \leftarrow M_2\}$. We have $\gamma_1[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \ @ \ \delta_i\big) \wedge \big(\bigwedge_{i,j} d_i \ @ \ d_{ij} \ @ \ \delta_j\big)$. Lemma B.6 shows that we can construct γ'_N and a δ'_i for all i such that $\gamma'_N[x \mapsto d_N] = \gamma_N$, and $\delta'_i[x \mapsto d^0_i] = \delta_i$ for all i and $\gamma_1 = \gamma'_N \wedge \big(\bigwedge_i d_i \ @ \ \delta'_i\big) \wedge \big(\bigwedge_{i,j} d_i \ @ \ d_{ij} \ @ \ \delta'_j\big)$, with $d_N = \gamma_N(x)$ and $d^0_i = \delta_i(x)$ for each i. Thus, the two premises can be rewritten as follows:

$$\begin{gathered}
\Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N : \tau \ / \ \gamma'_N[\ldots\ x_j \mapsto d_j\ \ldots][x \mapsto d_N] \\
\text{for all } i,\ \Gamma' + \{\ldots\ x_j : \tau_j\ \ldots\} \vdash N_i : \tau_i \ / \ \delta'_i[\ldots\ x_j \mapsto d_{ij}\ \ldots][x \mapsto d^0_i]
\end{gathered}$$

By induction hypothesis, it follows that

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

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

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

According to lemma B.1, the degree environment above is equal to

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

To obtain the expected result, it suffices to prove that this degree environment is equal to γ_0. Since

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

we know that

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

Therefore, $d = d_N \wedge \big(\bigwedge_i d_i @ d^0_i\big) \wedge \big(\bigwedge_{i,j} d_i @ d_{ij} @ d^0_j\big)$. It follows that

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

This completes the proof. □

We now extend the previous lemma to the case of parallel substitution, exploiting the fact that $M\{\ldots\ x_i \leftarrow M_i\ \ldots\}$ is equal to $M\{x_1 \leftarrow y_1\}\ \ldots\ \{x_n \leftarrow y_n\}\{y_1 \leftarrow M_1\}\ \ldots\ \{y_n \leftarrow M_n\}$, where the y_i are fresh. To support this reduction, we first show the stability of the typing judgement under substitution of one variable by a fresh variable.

LEMMA B.11. *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. □

LEMMA B.12. *(Parallel substitution.) Assume* $\Gamma + \{\dots\ x_i : \tau_i\ \dots\} \vdash M : \tau \ / \ \gamma_M[\dots\ x_i \mapsto d_i\ \dots]$, *and for all* $j \in \{1 \dots n\}$, $\Gamma \vdash M_j : \tau_j \ / \ \gamma_j$ *with for all* i, j, $x_i \notin FV(M_j) \cup dom(\gamma_j)$. *Then,* $\Gamma \vdash M\{\dots\ x_i \leftarrow M_i\ \dots\} : \tau \ / \ \gamma_M \wedge \bigwedge_i d_i \mathbin{@} \gamma_i$.

PROOF. Write $M\{\dots\ x_i \leftarrow M_i\ \dots\}$ as $M\{x_1 \leftarrow y_1\}\ \dots\ \{x_n \leftarrow y_n\}\{y_1 \leftarrow M_1\}\ \dots\ \{y_n \leftarrow M_n\}$ where the y_i are fresh. We first apply lemma B.11 n times to obtain $\Gamma + \{\dots\ y_i : \tau_i\ \dots\} \vdash M\{x_1 \leftarrow y_1\}\ \dots\ \{x_n \leftarrow y_n\} : \tau \ / \ \gamma_M[\dots\ y_i \mapsto d_i\ \dots]$. We then apply lemma B.10 n times again, successively using the n typing hypotheses for the M_i. This leads to the desired judgment. □

B.4 Substitution by a variable

We now state and prove a stronger variant of lemma B.10 for the case where we substitute a variable by another variable. This alternate substitution lemma is distinct from lemma B.11: 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 B.13. *(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 write $\Gamma' = \Gamma + \{x \mapsto \tau'\}$ and $M' = M\{x \leftarrow y\}$ and proceed by induction on the typing derivation of M and case analysis on the last typing rule used.

Rule (var) We distinguish the three sub-cases $M = x$, $M = y$, and $M = z$ with $z \neq x$ and $z \neq y$. All three cases are straightforward.

Rule (abstr), $M = \lambda z.M_1$ where z is fresh. By typing hypothesis, 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

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

Applying the 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

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

We conclude $\Gamma \vdash M\{x \leftarrow y\} : \tau \ / \ \gamma \wedge (y \mapsto d)$ by rule (abstr).

Rule (app), $M = M_1\ M_2$. The typing hypothesis entails $\Gamma' \vdash M_1 : \tau' \xrightarrow{d_0} \tau \ / \ \gamma_1$ and $\Gamma' \vdash M_2 : \tau' \ / \ \gamma_2$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$. Take $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$ and $\gamma'_2 = \gamma_2[x \mapsto \infty]$. These degree environments enjoy the following properties:

$$\gamma_1 = \gamma'_1[x \mapsto \gamma_1(x)] \qquad \gamma_2 = \gamma'_2[x \mapsto \gamma_2(x)] \qquad \gamma = (\gamma'_1 - 1) \wedge d_0 \mathbin{@} \gamma'_2.$$

By induction hypothesis, we can derive

$$\frac{\begin{array}{c}\Gamma \vdash M_1\{x \leftarrow y\} : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma'_1 \wedge (y \mapsto \gamma_1(x)) \\ \Gamma \vdash M_2\{x \leftarrow y\} : \tau'' \;/\; \gamma'_2 \wedge (y \mapsto \gamma_2(x))\end{array}}{\Gamma \vdash M' : \tau \;/\; (\gamma'_1 - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge d_0 \mathbin{@} (\gamma'_2 \wedge (y \mapsto \gamma_2(x)))}$$

The degree environment in the conclusion is equal to

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

The desired result follows.

Rule (appvar), $M = M_1\ z$ We have $\Gamma' \vdash M_1 : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_1$ and $\Gamma'(z) = \tau''$ and $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge (z \mapsto d_0)$. We consider the two cases $z = x$ and $z \neq x$ separately.

Case $z = x$. In this case, $\tau' = \tau''$. Consider $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. We have $\gamma'_1 - 1 = \gamma$ and $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$. By induction hypothesis, we obtain

$$\Gamma \vdash M_1\{x \leftarrow y\} : \tau' \xrightarrow{d_0} \tau \;/\; \gamma'_1 \wedge (y \mapsto \gamma_1(x)).$$

Since $\Gamma(y) = \tau'$, rule (appvar) concludes

$$\Gamma \vdash M' : \tau \;/\; (\gamma'_1 - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge (y \mapsto d_0).$$

The degree environment in this conclusion is equal to $(\gamma'_1 - 1) \wedge (y \mapsto ((\gamma_1(x) - 1) \wedge d_0))$, that is, $\gamma \wedge (y \mapsto d)$. This is the expected result.

Case $z \neq x$. Define $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. 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 obtain

$$\Gamma \vdash M_1\{x \leftarrow y\} : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma'_1 \wedge (y \mapsto \gamma_1(x)).$$

Since $\Gamma(z) = \tau''$, we derive by rule (appvar)

$$\Gamma \vdash M' : \tau \;/\; (\gamma'_1 - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge (z \mapsto d_0).$$

The latter degree environment is equal to $\gamma \wedge (y \mapsto (\gamma_1(x) - 1))$, that is, $\gamma \wedge (y \mapsto d)$, as required to establish the result.

Rule (rec), $M = \textbf{let rec}\ \ldots\ x_i = M_i\ \ldots\ \textbf{in}\ N$ where the x_i are fresh. The premises of rule (rec) hold:

$$\begin{array}{c}\Gamma' + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash M_j : \tau_j \;/\; \gamma_j[\ldots\ x_j \mapsto d_{ji}\ \ldots] \text{ for all } j \\ \Gamma' + \{\ldots\ x_i : \tau_i\ \ldots\} \vdash N : \tau \;/\; \gamma_N[\ldots\ x_i \mapsto d_i\ \ldots] \\ \text{for all } i, j,\ d_{ij} \geq 1 \\ \text{for all } i, j, k,\ d_{ik} \leq d_{ij} \mathbin{@} d_{jk}.\end{array}$$

Moreover, $\gamma[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$. By lemma B.6, we can construct γ'_N and γ'_i for each i satisfying the following conditions: $\gamma = \gamma'_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big)$, $\gamma_N = \gamma'_N[x \mapsto d_N]$, and for all i, $\gamma_i = \gamma'_i[x \mapsto d'_i]$, with $d_N = \gamma_N(x)$ and for all i, $d'_i = \gamma_i(x)$. Applying the induction hypothesis, we obtain derivations for the following judgments:

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

From these premises, rule (rec) derives $\Gamma \vdash M' : \tau \ /\ \gamma'$, where

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

This concludes the proof. □

B.5 Soundness

The soundness of λ_B's type system (theorem 5.1) is, as usual, a corollary of two properties: subject reduction (lemma B.15) and progress (lemma B.16). We start with a technical lemma on recursive definitions arising from the reduction of a **let rec** term.

LEMMA B.14. *Assume* $\Gamma + \{\dots\ x_i : \tau_i\ \dots\} \vdash M_j : \tau_j \ /\ \gamma_j[\dots\ x_i \mapsto d_{ji}\ \dots]$ *for all* $j \in \{1 \dots n\}$. *Further assume that for all* i, j, $d_{ij} \geq 1$ *and for all* i, j, k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$. *Then, for any* $i_0 \in \{1 \dots n\}$,

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

PROOF. By application of rule (rec), we obtain

$$\Gamma \vdash \textbf{let rec}\ \dots\ x_i = M_i\ \dots\ \textbf{in}\ M_{i_0} : \tau_{i_0} \ /\ \gamma_{i_0} \wedge \bigwedge_{i,j} d_{i_0 i} \mathbin{@} d_{ij} \mathbin{@} \gamma_j \wedge \bigwedge_i d_{i_0 i} \mathbin{@} \gamma_i.$$

Since $d_{i_0 j} \leq d_{i_0 i} \mathbin{@} d_{ij}$, we have $d_{i_0 j} \mathbin{@} \gamma_j \leq d_{i_0 i} \mathbin{@} d_{ij} \mathbin{@} \gamma_j$. Thus,

$$\bigwedge_{i,j} d_{i_0 i} \mathbin{@} d_{ij} \mathbin{@} \gamma_j \wedge \bigwedge_i d_{i_0 i} \mathbin{@} \gamma_i = \bigwedge_i d_{i_0 i} \mathbin{@} \gamma_i$$

and the expected result follows. □

LEMMA B.15. *(Subject reduction.) If* $\Gamma \vdash M : \tau \ /\ \gamma$ *and* $M \to M'$, *then* $\Gamma \vdash M' : \tau \ /\ \gamma$.

PROOF. The proof is by case analysis on the reduction rule used.

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

In the (appvar) case, we have $v = y$. We rename x if necessary to ensure $x \neq y$. The typing derivation for M is of the following form

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

Moreover, $\gamma = (\gamma_0 - 1) \wedge (y \mapsto d)$ and $M' = M_1\{x \leftarrow y\}$. By lemma B.13, we have

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

which is the expected result.

In the (app) case, the typing derivation for M is

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

Moreover, $M' = M_1\{x \leftarrow v\}$ and $\gamma = (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$. By lemma B.10, it follows that $\Gamma \vdash M' : \tau \;/\; \gamma$, as expected.

Reduction rule (mutrec), $M = \textbf{let rec} \;\ldots\; x_i = v_i \;\ldots\; \textbf{in}\; N$, where the x_i are fresh. We have $M' = M\{\ldots\; x_i \leftarrow M_i \;\ldots\}$ with, for all i, $M_i = \textbf{let rec} \;\ldots\; x_j = v_j \;\ldots\; \textbf{in}\; v_i$. By typing, we have

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

By lemma B.14, it follows that

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

By lemma B.12, 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 identical to the expected result

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

Reduction rule (context), $M = \mathbb{E}\,[M_1]$, $M_1 \to M_1'$ and $M' = \mathbb{E}\,[M_1']$. The result follows by structural induction and case analysis over the context $\mathbb{E}$. The only point worth mentioning is that in the case $\mathbb{E} = v\ [\,]$ and the typing derivation ends with rule (appvar), then M_1 can only be a variable, and therefore cannot reduce. □

LEMMA B.16. *(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. The proof is a standard inductive argument on the typing derivation of M, and case analysis on the last typing rule used.

Rule (var). M is a variable, i.e. a value.

Rule (abstr). M is a λ-abstraction, i.e. a value.

Rule (app), $M = M_1 \ M_2$. We have $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \ / \ \gamma_1$ and $\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$. Moreover, $\gamma = (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$. Applying the induction hypothesis to M_1 and M_2, either both terms are values or at least one reduces. If M_1 reduces, M also reduces via the context $[] \ M_2$. If M_1 is a value and M_2 reduces, M also reduces via the context $M_1 \ []$. If both M_1 and M_2 are values, the type $\tau' \xrightarrow{d} \tau$ of M_1 guarantees that M_1 is either a variable or an abstraction. But M_1 cannot be a variable, because $\gamma \geq 1$ implies $\gamma_1 \geq 2$. Hence, M_1 is an abstraction and we can apply rule (beta) to reduce M.

Rule (appvar). Same reasoning as in the (app) case.

Rule (rec), $M = \textbf{let rec} \ \ldots \ x_i = M_i \ \ldots \ \textbf{in} \ N$. If all M_i are values, M reduces by rule (mutrec). Otherwise, M reduces via the rule (context). □

C. SOUNDNESS OF THE TRANSLATION

We now turn to proving the type soundness of the translation: the translation of a well-typed source term is a well-typed λ_B-term.

We start by stating three typing rules that are admissible in λ_B, and help type-check the terms arising from the translation scheme. We omit the proofs of admissibility, which are straightforward.

LEMMA C.1. *(Single **let rec**.) The following typing rule is admissible for the type system of λ_B.*

$$\frac{\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \ / \ \gamma_1[x \mapsto d] \qquad \Gamma + \{x \mapsto \tau'\} \vdash N : \tau' \ / \ \gamma_2[x \mapsto d'] \qquad d' \geq 1}{\Gamma \vdash \textbf{let rec} \ x = N \ \textbf{in} \ M : \tau \ / \ \gamma_1 \wedge d \mathbin{@} \gamma_2}$$

LEMMA C.2. *(Multiple abstractions.) The following typing rule is admissible for the type system of λ_B.*

$$\frac{\Gamma + \{\ldots \ x_i : \tau_i \ \ldots\} \vdash M : \tau \ / \ (\gamma - n)[\ldots \ x_i \mapsto d_i \ \ldots]}{\Gamma \vdash \vec{\lambda}(x_1, \ldots, x_n).M : \tau_1 \xrightarrow{d_1+(n-1)} \tau_2 \xrightarrow{d_2+(n-2)} \ldots \tau_n \xrightarrow{d_n} \tau \ / \ \gamma}$$

LEMMA C.3. *(Multiple applications.) The following typing rule is admissible for the type system of λ_B.*

$$\frac{\Gamma \vdash M : \tau_1 \xrightarrow{d_1+(n-1)} \tau_2 \xrightarrow{d_2+(n-2)} \ldots \tau_n \xrightarrow{d_n} \tau \ / \ \gamma \qquad \Gamma(x_i) = \tau_i \ \textit{for} \ i = 1, \ldots, n}{\Gamma \vdash M(x_1, \ldots, x_n) : \tau \ / \ (\gamma - n) \wedge (\ldots \ x_i \mapsto d_i \ \ldots)}$$

We now prove two technical lemmas on the typing of sub-expressions that occur when translating the close and freeze operators.

LEMMA C.4. *(Translation of close.) Assume* $\Gamma \vdash e : [\![\{\mathcal{I};\mathcal{O};\mathcal{D}\}]\!] \ / \ d^o(\Gamma)$*. Let* $X_1,\ldots,X_n$ *be names such that* $\overline{X_i} \notin dom(\Gamma)$ *and* $\mathcal{O}(X_i) = \mathcal{I}(X_i)$ *and* $\mathcal{D}(X_i,X_j) \neq 0$ *for* $i,j \in \{1,\ldots,n\}$*. Further assume that for all immediate predecessors* X *of one of the* X_i *in* $\mathcal{D}$*, either* X *is one of the* X_i*, or* $\Gamma(\overline{X}) = \mathcal{I}(X)$*. Let* M *be an expression and* τ *be a type such that* $\Gamma' \vdash M : \tau \ / \ d^o(\Gamma')$*, where* $\Gamma' = \Gamma + \{\overline{X_1} : \mathcal{O}(X_1),\ldots,\overline{X_n} : \mathcal{O}(X_n)\}$*. Then,*

$$\Gamma \vdash \textbf{let rec } \ldots \overline{X_i} = e.X_i \ \overline{\mathcal{D}^{-1}(X_i)} \ldots \textbf{ in } M : \tau \ / \ d^o(\Gamma)$$

PROOF. By definition of the translation of a mixin signature, and the hypotheses on Γ, the conditions of lemma C.3 are met, and we obtain

$$\Gamma' \vdash e.X_i \ \overline{\mathcal{D}^{-1}(X_i)} : \mathcal{O}(X_i) \ / \ d^o(\Gamma) \wedge (\overline{X} \mapsto D(X,X_i) \mid X \in \mathcal{D}^{-1}(X_i)).$$

Since $\overline{X_j} \notin dom(\Gamma)$ for all j, the degree environment above is pointwise greater or equal to $d^o(\Gamma)[\overline{X_j} \mapsto D(X_j,X_i) \mid j \in \{1,\ldots,n\}]$. Thus, by lemma B.7, it follows that

$$\Gamma' \vdash e.X_i \ \overline{\mathcal{D}^{-1}(X_i)} : \mathcal{O}(X_i) \ / \ d^o(\Gamma)[\overline{X_j} \mapsto D(X_j,X_i) \mid j \in \{1,\ldots,n\}].$$

Moreover, $D(X_j,X_i) \in \{1,\infty\}$ for all i and j. Hence, the premises of the (rec) typing rule are met. Applying the weakening lemma B.7 to its conclusion, we obtain the desired result. □

LEMMA C.5. *(Translation of freeze.) Assume* $\Gamma \vdash e : [\![\{\mathcal{I};\mathcal{O};\mathcal{D}\}]\!] / d^o(\Gamma)$*, where* e *is a variable distinct from* $\overline{X}$ *for all names* X*. Let* X *be a name such that* $\mathcal{I}(X) = \mathcal{O}(X)$*. Write* $\mathcal{D}' = D\,!\,X$ *and* $\mathcal{I}' = \mathcal{I}_{\setminus X}$*. Then, for all names* $Y \in dom(\mathcal{O})$*, if* $X \notin \mathcal{D}^{-1}(Y)$ *we have*

$$\Gamma \vdash e.Y : [\![\mathcal{O}(Y)]\!]_{Y,\mathcal{D}',\mathcal{I}'} \ / \ d^o(\Gamma)$$

and if $X \in \mathcal{D}^{-1}(Y)$*, we have*

$$\Gamma \vdash \vec{\lambda}\,\overline{\mathcal{D}'^{-1}(Y)}.\textbf{let rec } \overline{X} = e.X \ \overline{\mathcal{D}^{-1}(X)} \textbf{ in } e.Y \ \overline{\mathcal{D}^{-1}(Y)} : [\![\mathcal{O}(Y)]\!]_{Y,\mathcal{D}',\mathcal{I}'} \ / \ d^o(\Gamma)$$

PROOF. Recall the definition of $\mathcal{D}'$:

$$\mathcal{D}' = \mathcal{D}\,!\,X = (\mathcal{D} \cup \mathcal{D}_{around}) \setminus \mathcal{D}_{remove}$$

where $\mathcal{D}_{around} = \{Z \xrightarrow{\chi_1' \wedge \chi_2'} Y \mid (Z \xrightarrow{\chi_1} X) \in \mathcal{D},\ (X \xrightarrow{\chi_2} Y) \in \mathcal{D}\}$ and $\mathcal{D}_{remove} = \{X \xrightarrow{\chi} Y \mid Y \in \mathit{Names},\ \chi \in \{0,1\}\}$.

Thus, in the case $X \notin \mathcal{D}^{-1}(Y)$, no edges leading to Y are added nor removed. Hence, $\mathcal{D}'^{-1}(Y) = \mathcal{D}^{-1}(Y)$, which implies $[\![\mathcal{O}(X)]\!]_{X,\mathcal{D}!X,\mathcal{I}_{\setminus X}} = [\![\mathcal{O}(X)]\!]_{X,\mathcal{D},\mathcal{I}}$ and the expected result.

Consider now the case $X \in \mathcal{D}^{-1}(Y)$. We have $\mathcal{D}'^{-1}(Y) = (\mathcal{D}^{-1}(Y) \cup \mathcal{D}^{-1}(X)) \setminus \{X\}$. Define $\Gamma' = \Gamma + \{\overline{Z} : [\![\mathcal{I}(Z)]\!] \mid Z \in \mathcal{D}^{-1}(Y)\}$. By lemma C.3, and using the fact that e is not one of the $\overline{Z}$, it follows that

$$\Gamma' \vdash e.X \ \overline{\mathcal{D}^{-1}(X)} : [\![\mathcal{O}X]\!] \ / \ \{e \mapsto 0; \overline{Z} \mapsto \mathcal{D}(Z,X) \mid Z \in \mathcal{D}^{-1}(X)\}$$

and

$$\Gamma' + \{\overline{X} : [\![\mathcal{I}(X)]\!]\} \vdash e.Y \ \overline{\mathcal{D}^{-1}(Y)} : [\![\mathcal{O}Y]\!] \ / \ \{e \mapsto 0; \overline{Z} \mapsto \mathcal{D}(Z,Y) \mid Z \in \mathcal{D}^{-1}(Y)\}.$$

Notice that $\mathcal{D}(X, X) \geq 1$, because otherwise the graph $\mathcal{D}$ would not be safe, making the signature $\{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$ ill-formed. In addition, $\mathcal{O}(X) = \mathcal{I}(X)$. The conditions of lemma C.1 are therefore met, and we obtain $\Gamma' \vdash \textbf{let rec } \overline{X} = e.X\ \overline{\mathcal{D}^{-1}(X)}$ **in** $e.Y\ \overline{\mathcal{D}^{-1}(Y)} : [\![\mathcal{O}(Y)]\!] \ / \ \gamma$, where

$$\begin{aligned}\gamma \ = \ &\{e \mapsto 0; \overline{Z} \mapsto \mathcal{D}(Z, X) \mid Z \neq X, Z \in \mathcal{D}^{-1}(X)\} \\ &\wedge \{e \mapsto 0; \overline{Z} \mapsto \mathcal{D}(Z, Y) \mid Z \neq X, Z \in \mathcal{D}^{-1}(Y)\}\end{aligned}$$

By definition of $\mathcal{D}' = \mathcal{D} \,!\, X$, γ is equal to $\{e \mapsto 0; \overline{Z} \mapsto \mathcal{D}'(Z, Y) \mid Z \in \mathcal{D}'^{-1}(Y)\}$. Applying lemma C.2, we obtain

$$\Gamma \vdash \vec{\lambda}\, \overline{\mathcal{D}'^{-1}(Y)}.\ \textbf{let rec}\ \overline{X} = e.X\ \overline{\mathcal{D}^{-1}(X)}\ \textbf{in}\ e.Y\ \overline{\mathcal{D}^{-1}(Y)} : [\![\mathcal{O}(X)]\!]_{X,\mathcal{D}',\mathcal{I}'} \ / \ \{e \mapsto 0\}$$

which implies the desired result by weakening. □

THEOREM C.6. *(Soundness of the translation.) If* $\Gamma \vdash E : \mathcal{T}$, *then* $[\![\Gamma]\!] \vdash [\![\overline{E}]\!] : [\![\mathcal{T}]\!] \ / \ d^o(\Gamma) + \mathit{IsRec}(E)$.

PROOF. The proof is by structural induction on E, and case analysis on E.

Function abstraction: $E = \lambda x.C$ and $\mathcal{T} = \tau_1 \to \tau_2$. By induction hypothesis, $[\![\Gamma]\!] + \{x : \tau_1\} \vdash [\![C]\!] : \tau_2 \ / \ d^o(\Gamma)[x \mapsto 0] + \mathit{IsRec}(C)$. Applying the degree weakening lemma if $\mathit{IsRec}(C)$ is not zero, we obtain $[\![\Gamma]\!] + \{x : \tau_1\} \vdash [\![C]\!] : \tau_2 \ / \ d^o(\Gamma)[x \mapsto 0]$. From this, the (abstr) typing rule shows that $[\![\Gamma]\!] \vdash [\![\lambda x.C]\!] : \tau_1 \xrightarrow{0} \tau_2 \ / \ d^o(\Gamma) + 1$, which is the expected result since $\mathit{IsRec}(\lambda x.C) = 1$.

Other core language constructs: the result follows immediately from the induction hypothesis, since $\mathit{IsRec}(E) = 0$ in these cases.

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

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 $[\![\Gamma]\!] + [\![\mathcal{I} \circ \iota]\!] \vdash [\![\overline{E_i}]\!] : [\![\mathcal{T}_i]\!] \ / \ d^o(\Gamma + \mathcal{I} \circ \iota) + \chi_i$.

Notice that $FV([\![\overline{E_i}]\!]) = FV(E_i)$ and $FV(E_i) \cap \mathit{dom}(\iota) = \iota^{-1}(\mathcal{D}^{-1}(X_i))$. We can therefore apply lemma C.2 and weakening lemmas B.8 and B.9 to eliminate variables of $\mathit{dom}(\iota)$ 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

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

Moreover, we have $[\![\mathcal{T}_i]\!]_{X_i,\mathcal{D},\mathcal{I}} = [\![\mathcal{T}''_1]\!] \xrightarrow{\chi_i + (n-1)} \ldots [\![\mathcal{T}''_n]\!] \xrightarrow{\chi_i} [\![\mathcal{T}_i]\!]$ as a consequence of $\mathcal{D}(Z_k, X_i) = \nu(\iota^{-1}(Z_k),\ E_i) = \mathit{IsRec}(E_i) = \chi_i$. The desired result follows.

Closing: $E = \texttt{close}(E')$ and $\mathcal{T} = \{\mathcal{I}; \mathcal{O}; \mathcal{D}\}$. We apply lemma C.4 repeatedly to each **let rec** group in the translation, starting with the innermost one. Since the **let rec** are generated following a serialisation of the graph $\mathcal{D}$, all free variables in a **let rec** are bound earlier, and dependencies between the variables bound in the same **let rec** cannot have degree 0 (otherwise the graph $\mathcal{D}$ would not be safe, and $\mathcal{T}$ would be ill-formed). The expected result follows.

Freezing: $E = E_1 \,!\, X$. The result follows from the induction hypothesis applied to E_1, and lemma C.5 applied to each component of the record generated by the translation.

Delete: $E = E_1 \setminus X$. The result follows immediately from the induction hypothesis applied to E_1.

Renaming: $E = E_1[X \leftarrow Y]$. We apply the induction hypothesis to E_1, then use lemmas C.2 and C.3 to handle the rearrangement of the parameters of the record components. □