Rigid mixin modules.

Tom Hirschowitz

Laboratoire de l'Informatique du Parallélisme

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

Rigid mixin modules

Tom Hirschowitz

October 2003

Research Report N° 2003-46

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

Rigid mixin modules

Tom Hirschowitz

October 2003

Abstract

Mixin modules are a notion of modules that allows cross-module recursion and late binding, two features missing in ML-style modules. They have been well defined in a call-by-name setting, but in a call-by-value setting, they tend to conflict with the usual static restrictions on recursive definitions. Moreover, the semantics of instantiation has to specify an order of evaluation, which involves a difficult design choice. Previous proposals [12, 14] rely on the dependencies between components to compute a valid order of evaluation. In such systems, mixin module types must carry some information about the dependencies between their components, which makes them rather impractical. In this paper, we propose a new design for mixin modules in a call-by-value setting, which avoids this problem. The formalism we obtain is much simpler than previous notions of mixin modules, although slightly less powerful.

Keywords: Programming languages, semantics, typing, modularity, mixin modules.

Résumé

Les modules mixins sont une notion de modules qui permet la récursion entre modules et la liaison tardive, deux traits manquant aux modules de ML. Néanmoins, leur définition est plus aisée en appel par nom. Dans un contexte d'appel par valeur, elle pose des problèmes de définitions récursives illégales et d'ordre d'évaluation au moment de l'instantiation. Des travaux précédents ont proposé de s'appuyer sur les dépendances entre définitions pour calculer un ordre d'évaluation valide. Dans ces systèmes, les types de modules mixins doivent contenir de l'information sur les dépendances entre leurs composantes, ce qui les rend inadaptés. Dans ce papier, nous proposons une nouvelle définition des modules mixins en appel par valeur, qui évite ce problème. Le formalisme obtenu est beaucoup plus simple, bien que légèrement moins puissant.

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

1 Introduction

1.1 The problem

Modern languages provide a collection of features dedicated to modularity, called module systems, which offer powerful abstractions and static verifications, leading to fewer errors, while still allowing the construction of rich libraries of modules. The ML module system [18, 19, 10, 15], remains one of the most expressive. Nevertheless, this system is weak on at least two important points.

(Mutual recursion) Mutually recursive definitions cannot be split across separate modules, which hinders modularization in several cases [8, 6].

(Modifiability) Once a module is defined, the language does not propose any mechanism for modifying it. Instead, one has to copy the code manually, and create a new module from scratch.

Our long-term goal is to devise a module system for ML featuring both mutual recursion and modifiability, without loosing other important features of current ML modules, such as parameterization, efficiency, separate compilation, and data abstraction.

As a starting point, we notice that the two features we want to bring to ML modules are notably provided by class-based, object-oriented languages. Open recursion and abstract (or virtual) methods naturally allow one to define mutually recursive methods across classes, which are later merged together by inheritance. Furthermore, inheritance features overriding and late binding, which addresses the mentioned modifiability requirement.

So, why not simply adopt the object-oriented approach for module systems? Essentially, because the components of an object are more or less restricted to be functions. Instance variables and initializers[1] [16] slightly improve over this restriction, but remain cumbersome and error-prone. Instead, it should be possible to naturally interleave functional components with computational, possibly side-effective components using the previous ones. However, extending classes and objects this way raises the following important problems.

(Recursive definitions) Arbitrary recursive definitions can appear dynamically, because of inheritance. In most call-by-value languages, recursive definitions are statically restricted, in order to be more efficiently implementable [3, 13], and to avoid some ill-founded definitions. Obviously, our system should not force language designers to abandon these properties, and thus needs guards on recursive definitions, at the level of both static and dynamic semantics.

(Order of evaluation) In our system, classes will contain arbitrary, unevaluated definitions, whose evaluation will be triggered by instantiation. Because these definitions are arbitrary, the order in which they will be evaluated matters. For instance, in a class c defining $x = 0$ and $y = x + 1$, x must be evaluated before y. Thus, the semantics of instantiation must define an order of evaluation. Moreover, classes can be built by inheritance, so the semantics of inheritance must also take the order of definitions into account.

[1] Initializers are methods that are automatically called once at initialization time.

From the standpoint of dynamic semantics, restricting recursive definitions is simply done by syntactically constraining the set of terms. The second issue is more difficult, because it involves a design decision. From the standpoint of typing, the second issue reduces to the first one, since the existence of a valid order of evaluation is governed by the absence of invalid recursive definitions.

1.2 Flexible call-by-value mixin modules

Hirschowitz, Leroy, and Wells [12, 14] adopt the following approach, in their *MM* language of call-by-value mixin modules. Mixin modules contain unordered definitions. Only at instantiation does the system compute an order for them, according to their inter-dependencies [12, 14], and to programmer-supplied annotations that fix some bits of the final order [14]. This solution is very expressive w.r.t. code reuse, since components can be re-ordered according to the context. However, it appears somewhat heavy in some respects.

(Instantiation) In particular, instantiation is too costly, since it involves computing the strongly-connected components of a graph whose size is quadratic in the input term, plus a topological sort of the result.

(Type safety) When recursive definitions are guarded, typing mixin modules is difficult, because invalid recursive definitions can appear dynamically. In order to prevent this, the proposed solution [12] is to enrich mixin module types with some information about the dependencies between definitions. Unfortunately, this makes mixin module types heavy, and also over-specified. Indeed, the least change in the dependencies between components forces the type of the mixin module to change.

The first problem is not so annoying in the context of a module system: it only has to do with linking operations, and thus should not affect the overall efficiency of programs. The second problem makes the proposed language impractical without dedicated graph support.

1.3 Rigid mixin modules

In this paper, we propose a completely different approach, from scratch. We introduce *Mix*, a new language of call-by-value mixin modules, where mixin module components are ordered, in a rigid way. They can be defined either as single components (briefly called "singles") or as blocks of components. Blocks contain mutually recursive definitions, and are restricted to a certain class of values. Conversely, singles can contain arbitrary, non-recursive computations. Composition preserves the order of both of its arguments, and instantiation straightforwardly translates its argument into a module.

With respect to side effects, annotations are no longer needed, since side effects always respect the syntactic order. Moreover, instantiation is less costly than in *MM*, since it runs in $O(n \log n)$, where n is the size of the input. Concerning typing, mixin module types have the same structure as mixin modules themselves: they are sequences of specifications, which can be either singles or blocks. Hence, they avoid the use of explicit graphs, which improves over *MM*. Compared to ML module types, the only differences are that the order matters and that mutually recursive specifications must be explicitly grouped together.

Finally, the meta theory of *Mix* is much simpler than the one of *MM*, which makes it more likely to scale up to a full-featured language like ML. The price to pay for these advantages is a reduced flexibility. Deciding whether *Mix* is still expressive enough is outside the scope of this paper.

The rest of the paper is organized as follows. Section 2 presents an informal overview of *Mix* by example. Section 3 formally defines *Mix* and its dynamic semantics. Section 4 defines a sound type system for *Mix*. Finally, sections 5 and 6 review related and future work, respectively. The proofs are relegated in appendix for readability.

2 Intuitions

As a simplistic introductory example, consider a program that defines two mutually recursive functions for testing whether an integer is even or odd, and then tests whether 56 is even, and whether it is odd. Assume now that it is conceptually obvious that everything concerning oddity must go into one program fragment, and everything concerning evenness must go into another, clearly distinct fragment. Here is how this can be done in an informal programming language based on *Mix*, with a syntax mimicking OCaml [16].

First, define two mixin modules `Even` and `Odd` as follows.

```
mixin Even = mix
  let rec ? odd : int -> bool
          ! even x = x = 0 or odd (x-1)
  let ! even56 = even 56
end

mixin Odd = mix
  let rec ? even : int -> bool
          ! odd x = x > 0 and even (x-1)
  let ! odd56 = odd 56
end
```

Each of these mixin modules declares the missing function (marking it with `?`) and defines the other one (marking it with `!`), inside a `let rec` which delimits a recursive block. Then, outside of this block, each mixin module performs one computation.

In order to link them, and obtain the desired complete mixin module, one composes `Even` and `Odd`, by writing `mixin OpenNat = Odd >> Even`. This has the effect of somehow passing `Odd` through `Even`, with `Even` acting as a filter, stopping the components of `Odd` when they match one of its own components. This filtering is governed by some rules: the components of `Odd` go through `Even` together, until one of them, say component c, matches some component of `Even`. Then, the components of `Odd` defined to the left of c are stuck at the current point. The other components continue their way through `Even`. Additionally, when two components match, they are merged into a single component.

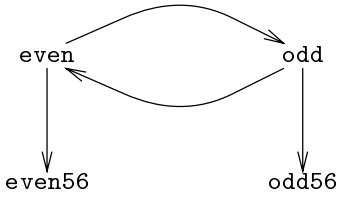
In our example, `odd` and `even` both stop at `Even`'s recursive block mentioning them, so the two recursive blocks are merged. Further, `odd56` continues until the end of `Even`. The obtained mixin module is thus equivalent to

```
mixin OpenNat = mix
  let rec ! even x = x = 0 or odd (x-1)
          ! odd x = x > 0 and even (x-1)
  let ! even56 = even 56
  let ! odd56 = odd 56
end
```

Notice that composition is definitely asymmetric. This mixin module remains yet to be instantiated, in order to trigger its evaluation. This is done by writing `module Nat = close OpenNat`. This makes it into a module, equivalent in OCaml syntax to

```
module Nat = struct
  let rec even x = x = 0 or odd (x-1)
          odd x = x > 0 and even (x-1)
  let even56 = even 56
  let odd56 = odd 56
end
```

which evaluates to the desired result. For comparison, in *MM*, the final evaluation order would be computed upon instantiation, rather than upon composition. It would involve a topological sort of the strongly-connected components of the following dependency graph.



Incidentally, in order to ensure that `even56` is evaluated before `odd56`, the definition of `odd56` should better explicitly state it.

3 The *Mix* language and its dynamic semantics

3.1 Syntax

Pre-terms Figure 1 defines the set of *pre-terms* of *Mix*. It distinguishes names X from variables x, following Harper and Lillibridge [10]. It includes a standard record construct $\{s\}$, where $s ::= (X_1 = e_1 \ldots X_n = e_n)$ and selection $e.X$. It features two constructs for value binding, letrec for mutually recursive definitions, and let for single, non-recursive definitions. Finally, the language provides four mixin module constructs. Basic mixin modules, called *structures*, consist of *mixtures* $m = (c_1 \ldots c_n)$, wich are lists of *components*. A component c is either a *single*, or a *block*. A single u is either a named *declaration* $X \triangleright x = \bullet$, or a *definition* $L \triangleright x = e$, where L is a *label*. Labels can be names or the special *anonymous* label, written _, in which case the definition is also said anonymous. Finally, a block q is a list of singles. The other constructs are composition ($e_1 \gg e_2$), instantiation (close e), and deletion of a name X, written ($e_{|-X}$).

x	$\in$	*Vars*	Variable
X	$\in$	*Names*	Name
L	$\in$	$\mathit{Names} \cup \{\_\}$	Label
Expression:			
e	$::=$	x	Variable
	$\mid$	$\{X_1 = e_1 \ldots X_n = e_n\}$	Record
	$\mid$	$e.X$	Selection
	$\mid$	letrec $x_1 = e_1 \ldots x_n = e_n$ in e	letrec
	$\mid$	let $x = e_1$ in e_2	let
	$\mid$	$\langle c_1 \ldots c_n \rangle$	Structure
	$\mid$	$e_1 \gg e_2$	Composition
	$\mid$	close e	Instantiation
	$\mid$	$e_{\mid -X}$	Deletion
Definition:			
c	$::=$	u	Single definition
	$\mid$	$[u_1 \ldots u_n]$	Block
Single definition:			
u	$::=$	$L \rhd x = e \mid X \rhd x = \bullet$	

Figure 1: Syntax

Terms Proper terms of the language are defined by restricting the set of pre-terms, as follows.

Definition 1 (Terms)
A term *of Mix is a pre-term such that: records do not define the same name twice ; bindings do not define the same variable twice ; mixtures define neither the same name twice nor the same variable twice ; and* letrec *definitions and definitions in blocks belong to the set RecExp of* valid recursive definitions*, which is a parameter of the system, but must be included in the set of values and stable under substitution.*

In terms, records, bindings and mixtures are often considered as finite maps from names to terms, variables to terms, and pairs of a label and a variable to terms or $\bullet$, respectively. From this standpoint, the domain of a mixture, restricted to pairs of a name and a variable, can in turn be seen as an injective finite map from names to variables. The domain of such finite maps is denoted by *dom*.

Terms are considered equivalent modulo proper renaming of bound variables and modulo the order in blocks and bindings. We denote by $DV(m)$ and $DV(b)$ the sets of variables defined by m and b, respectively, and by $DN(m)$ and $DN(s)$ the sets of names defined by m and s, respectively.

3.2 Dynamic semantics

The semantics of *Mix* is defined as a reduction relation on pre-terms in figure 2, using notions defined in figure 3. It is shown to be well-defined and stable over terms below. To begin with, we define *Mix* values v by $v ::= \{s^v\} \mid \langle m \rangle$, where $s^v ::= (X_1 = v_1 \ldots X_n = v_n)$. Then, figure 3 defines *evaluation contexts*, which

$$
\begin{array}{rcll}
\langle m_1\rangle \gg \langle m_2\rangle & \to & \langle Add(m_1, m_2, \varepsilon)\rangle \text{ if } m_1 \fatsemi m_2 & (\textsc{Compose}) \\
\mathsf{close}\ \langle m^c\rangle & \to & Bind(m^c, \{\lceil m^c \rceil\}) & (\textsc{Close}) \\
\langle m\rangle_{|-X} & \to & \langle Del(m, X)\rangle & (\textsc{Delete}) \\
\mathsf{letrec}\ b\ \mathsf{in}\ e & \to & \{x \mapsto \mathsf{letrec}\ b\ \mathsf{in}\ b(x) \mid x \in dom(b)\}(e) & (\textsc{LetRec}) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \to & \{x \mapsto v\}(e) & (\textsc{Let}) \\
\{s^v\}.X & \to & s^v(X) & (\textsc{Select}) \\
\mathbb{E}[e] & \to & \mathbb{E}[e'] \text{ if } e \to e' & (\textsc{Context})
\end{array}
$$

Figure 2: Reduction rules

enforce a deterministic, call-by-value strategy. We can now examine the rules, from the most interesting to the most standard.

Composition Rule Compose describes mixin module composition. In order to be composed, structures must be made *compatible* by α-conversion. Namely, we say that two mixtures m_1 and m_2 are compatible, and write $m_1 \fatsemi m_2$, iff for any $x \in (DV(m_1) \cup FV(m_1)) \cap (DV(m_2) \cup FV(m_2))$, there exists X such that $dom(m_1)(X) = dom(m_2)(X) = x$. This basically says that both mixtures agree on the names of variables.

Then, their composition $\langle m_1\rangle \gg \langle m_2\rangle$ is $Add(m_1, m_2, \varepsilon)$, where Add is defined by induction on m_1 by

$$
\begin{array}{rcl}
Add(\varepsilon, m_1, m_2) & = & m_1, m_2 \\
Add((m_1, c), m_2, m_3) & = & Add(m_1, m_2, (c, m_3)) \\
 & & \text{if } DN(c) \perp DN(m_2, m_3) \\
Add((m_1, c_1), (m_2^1, c_2, m_2^2), m_3) & = & Add(m_1, m_2^1, (c_1 \otimes c_2, m_2^2, m_3)) \\
 & & \text{if } DN(c_1) \perp DN(m_2^1, m_2^2) \text{ and } DN(c_1) \cap DN(c_2) \neq \emptyset
\end{array}
$$

Given three arguments m_1, m_2, m_3, Add roughly works as follows. If m_1 is empty, it returns the concatenation of m_2 and m_3. If the last component c of m_1 defines names that are not defined in m_2 or m_3, then c is pushed at the head of m_3. Finally, when the last component c_1 of m_1 defines a name that is also defined by some c_2 in m_2, so that $m_2 = (m_2^1, c_2, m_2^2)$, then the third argument becomes $(c_1 \otimes c_2, m_2^2, m_3)$, where $c_1 \otimes c_2$ is the *merge* of c_1 and c_2, which is defined by

$$
\begin{array}{rcll}
c_1 \otimes c_2 & = & c_2 \otimes c_1 & \\
(X \rhd x = \bullet) \otimes c & = & c & \text{if } dom(c)(X) = x \\
[q_1] \otimes [q_2] & = & [q_1, q_2] & \text{if } DN(q_1) \perp DN(q_2) \\
[X \rhd x = \bullet, q_1] \otimes [q_2] & = & [q_1] \otimes [q_2] & \text{if } dom(q_2)(X) = x
\end{array}
$$

This definition is not algorithmic, but uniquely defines the merging of two components, and an algorithm is easy to derive from it. Besides, it has the advantage of being relatively simple. Technically, as soon as a declaration is matched, it is removed, and when two blocks have no more common defined names, their merge is their union. Notice that initially, only components with common defined names are merged, but that the union takes place after all the common names have been reduced. This implements the behavior informally described in section 2.

$$
\begin{array}{rl}
\mathbb{E} ::= & \{s^v, X = \Box, s\} \mid \Box.X \\
& \mid \mathsf{let}\ x = \Box\ \mathsf{in}\ e \\
& \mid \Box \gg e \mid v \gg \Box \\
& \mid \mathsf{close}\ \Box \mid \Box_{|-X}
\end{array}
$$

Figure 3: Evaluation contexts

Instantiation Rule COMPOSE describes the instantiation of a *complete* structure. A mixture is said complete iff it does not contain declarations. We denote complete mixtures, components, singles, and blocks by m^c, c^c, u^c, and q^c, respectively.

Given a complete structure $\langle m^c \rangle$, instantiation first generates a series of bindings, following the structure of m^c, and then stores the results of named definitions in a record. Technically, $\mathsf{close}\ \langle m^c \rangle$ reduces to $\mathit{Bind}(m^c, \{\lceil m^c \rceil\})$, where $\lceil \cdot \rceil$ makes m^c into a record and *Bind* makes m^c into a binding:

$\lceil m^c \rceil$ is defined by

$$\lceil X \rhd x = e \rceil = (X = x) \qquad \text{and} \qquad \lceil \_ \rhd x = e \rceil = \varepsilon,$$

naturally extended to components and mixtures,

and $\mathit{Bind}(m^c, e)$ is defined inductively over m^c by

$$
\begin{array}{rcl}
\mathit{Bind}(\varepsilon, e) & = & e \\
\mathit{Bind}(([u^c{}_1 \ldots u^c{}_n], m^c), e) & = & \mathsf{letrec}\ \lfloor u^c{}_1 \rfloor \ldots \lfloor u^c{}_n \rfloor\ \mathsf{in}\ \mathit{Bind}(m^c, e) \\
\mathit{Bind}((u^c, m^c), e) & = & \mathsf{let}\ \lfloor u^c \rfloor\ \mathsf{in}\ \mathit{Bind}(m^c, e), \\
\text{with } \lfloor L \rhd x = e \rfloor & = & (x = e).
\end{array}
$$

For each component, *Bind* defines a letrec (if the component is a block) or a let (if the component is a single), by extracting bindings $x = e$ from singles $L \rhd (x = e)$.

Other rules Rule DELETE describes the action of the deletion operation. Given a structure $\langle m \rangle$, $\langle m \rangle_{|-X}$ reduces to $\langle \mathit{Del}(m, X) \rangle$, where $\mathit{Del}(m, X)$ denotes m, where any definition of the shape $X \rhd x = e$ is replaced with $X \rhd x = \bullet$.

The next two rules, LETREC, LET, handle value binding. The only non-obvious rule is LETREC, which enforces the following behavior. The rule applies when the considered binding is fully evaluated (which is always the case for terms). A pre-term $\mathsf{letrec}\ b\ \mathsf{in}\ e$, reduces to e, where each $x \in \mathit{dom}(b)$ is replaced with a kind of closure representing its definition, namely $\mathsf{letrec}\ b\ \mathsf{in}\ b(x)$. Notice the notation for capture-avoiding substitution.

Finally, rule SELECT defines record selection, and rule CONTEXT extends the rule to any evaluation context.

Reduction is well-defined and preserves term well-formedness, in the sense that the reduct of a term remains a term.

Proposition 1 (Compatibility)
If two structurally equivalent pre-terms e_1 and e_2 reduce to e_3 and e_4, respectively, then e_3 and e_4 are structurally equivalent.

Type:

$$\begin{array}{rcl} \tau \in \mathit{Types} & ::= & \{S\} \mid \langle C_1 \dots C_n \rangle \\ C & ::= & U \mid [U_1 \dots U_n] \\ U & ::= & \delta X : \tau \\ \delta & ::= & ! \mid ? \\ S & \in & \mathit{Names} \xrightarrow{\mathit{fin}} \mathit{Types} \end{array}$$

Environment:

$$\Gamma \in \mathit{Vars} \xrightarrow{\mathit{fin}} \mathit{Types}$$

Figure 4: Types

Proposition 2 (Stability)
If e is a term and $e \to e'$, then e' is a term.

4 Static semantics

We now define a sound type system for *Mix* terms. This means that the rules do not have to check that the considered expressions are proper terms, not just pre-terms. Types are defined in figure 4. An *Mix* type τ can be either a record type or a mixin module type. A mixin module type has the shape $\langle M \rangle$, where M is a *signature*. A signature is a list of *specifications* C, which can be either *single specifications* U or *block specifications* Q. A single specification has the shape $\delta X : \tau$ where δ is a flag indicating whether the considered name is a declaration of a definition. It can be either ?, for declarations, or !, for definitions. A block specification is a list of single specifications. Record types are finite maps from names to types. Types are identified modulo the order of specifications in blocks. Environments Γ are finite maps from variables to types. The disjoint union of two environments Γ_1 and Γ_2 is written $\Gamma_1 + \Gamma_2$.

Figure 5 presents our type system for *Mix*.

Structures and enriched specifications Let us begin with the typing of structures. Rule T-Struct simply delegates the typing of a structure $\langle m \rangle$ to the rules for typing mixtures. These rules basically give each component c an *enriched specification*, which is a specification, enriched with the corresponding variable. Formally, single enriched specifications have the shape $\delta L \rhd x : \tau$, and enriched block specifications are finite sets of these. Notably, this allows to type anonymous definitions (using enriched specifications like $\delta \_ \rhd x : \tau$), and also to recover a typing environment (namely $\{x \mapsto \tau\}$) for typing the next components. Enriched single specifications, block specifications, and signatures are denoted by U^e, Q^e, and M^e, respectively. Once the mixture m has been given such an enriched signature M^e, this result is converted to a proper signature $M = \mathit{Sig}(M^e)$, assigning to the structure $\langle m \rangle$ the type $\langle M \rangle$. The *Sig* function merely forgets variables and anonymous definitions of its argument: it is defined by straightforward extension of

$$\mathit{Sig}(\delta X \rhd x : \tau) = \delta X : \tau \qquad \mathit{Sig}(\delta \_ \rhd x : \tau) = \varepsilon.$$

Here is how mixtures are given such enriched signatures. By rule T-Some, a single definition $L \rhd x = e$ is given the enriched single specification $!L \rhd x : \tau$

Expressions

T-Struct
$$\frac{\Gamma \vdash c_1 \ldots c_n : M^e}{\Gamma \vdash \langle c_1 \ldots c_n \rangle : \langle Sig(M^e) \rangle}$$

T-Compose
$$\frac{\Gamma \vdash e_1 : \langle M_1 \rangle \qquad \Gamma \vdash e_2 : \langle M_2 \rangle}{\Gamma \vdash e_1 \gg e_2 : \langle Add(M_1, M_2, \varepsilon) \rangle}$$

T-Close
$$\frac{\Gamma \vdash e : \langle M^c \rangle}{\Gamma \vdash \mathsf{close}\ e : \{\lceil M^c \rceil\}}$$

T-Delete
$$\frac{\Gamma \vdash \langle m \rangle : \langle M \rangle}{\Gamma \vdash \langle m \rangle_{|-X} : \langle Del(M, X) \rangle}$$

T-Var
$$\Gamma \vdash x : \Gamma(x)$$

T-Record
$$\frac{dom(s) = dom(S) \qquad \forall X \in dom(s), \Gamma \vdash s(X) : S(X)}{\Gamma \vdash \{s\} : \{S\}}$$

T-Select
$$\frac{\Gamma \vdash e : \{S\}}{\Gamma \vdash e.X : S(X)}$$

T-LetRec
$$\frac{\Gamma + \Gamma_b \vdash b : \Gamma_b \qquad \Gamma + \Gamma_b \vdash e : \tau}{\Gamma \vdash \mathsf{letrec}\ b\ \mathsf{in}\ e : \tau}$$

T-Let
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Definitions

T-Some
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (L \rhd x = e) : (!L \rhd x : \tau)}$$

T-None
$$\Gamma \vdash (X \rhd x = \bullet) : (?X \rhd x : \tau)$$

Outputs and bindings

T-Empty
$$\Gamma \vdash \varepsilon : \varepsilon$$

T-Single
$$\frac{\Gamma \vdash u : U^e \qquad \Gamma + Env(U^e) \vdash m : M^e}{\Gamma \vdash (u, m) : (U^e, M^e)}$$

T-Block
$$\frac{\Gamma + \Gamma_q \vdash m : M^e \qquad \Gamma_q = \biguplus_{u \in q} Env({U^e}_u)}{\forall u \in q, u \in RecExp_? \text{ and } \Gamma + \Gamma_q \vdash u : {U^e}_u}$$
$$\Gamma \vdash ([q], m) : ([\biguplus_{u \in q} {U^e}_u], M^e)$$

T-Binding
$$\frac{dom(b) = dom(\Gamma_b) \qquad \forall x \in dom(b), b(x) \in RecExp \text{ and } \Gamma_b \vdash b(x) : \Gamma_b(x)}{\Gamma \vdash b : \Gamma_b}$$

Figure 5: Type system

if e has type τ. By rule T-NONE, a single declaration $X \rhd x = \bullet$ can be given any enriched specification of the shape $?X \rhd x : \tau$.

Given this, we can define the typing of mixtures. By rule T-EMPTY, an empty mixture is given the empty signature. By rule T-SINGLE, a mixture of the shape (u, m) is typed as follows. First, u is typed, yielding an enriched specification U^e. This U^e is made into an environment by the *Env* function from enriched signatures to environments. This function associates to any enriched single specification $\delta L \rhd x : \tau$ the finite map $\{x \mapsto \tau\}$, and is straightforwardly extended to signatures. The obtained environment is added to the current environment for typing the remaining components inductively, yielding an enriched signature M^e. The type of the whole mixture is (U^e, M^e).

By rule T-BLOCK, a mixture of the shape $([q], m)$ is typed as follows. An enriched single specification ${U^e}_u$ is first guessed for each single u of q. Then, they are converted into an environment $\Gamma_q = \biguplus_{u \in q} \mathit{Env}({U^e}_u)$.

This environment Γ_q is directly added to the current environment. Then, it is checked that each single u indeed has the enriched specification ${U^e}_u$. Additionally, it is checked that each single u of q is a valid recursive definition or a declaration. By abuse of notation, we write this $u \in \mathit{RecExp}_?$.

Finally, the mixture m is typed, yielding an enriched signature M^e, which is concatenated to $[\biguplus_{u \in q} {U^e}_u]$.

Composition The typing of composition closely follows the corresponding reduction rule, as defined by rule T-COMPOSE. The type of the composition of two mixin modules of types $\langle M_1 \rangle$ and $\langle M_2 \rangle$, respectively, is $\langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle$, where *Add* is defined by

$$\begin{array}{rcll} \mathit{Add}(\varepsilon, M_1, M_2) & = & M_1, M_2 & \\ \mathit{Add}((M_1, C), M_2, M_3) & = & \mathit{Add}(M_1, M_2, (C, M_3)) & \\ & & \text{if } \mathit{DN}(C) \perp \mathit{DN}(M_2, M_3) & \\ \mathit{Add}((M_1, C_1), (M_2^1, C_2, M_2^2), M_3) & = & \mathit{Add}(M_1, M_2^1, (C_1 \otimes C_2, M_2^2, M_3)) & \\ & & \text{if } \mathit{DN}(C_1) \perp \mathit{DN}(M_2^1, M_2^2, M_3) \text{ and } \mathit{DN}(C_1) \cap \mathit{DN}(C_2) \neq \emptyset & \end{array}$$

which does the same as *Add* on mixtures. The merging of two specifications is similarly defined by

$$\begin{array}{rcll} C_1 \otimes C_2 & = & C_2 \otimes C_1 & \\ (?X : \tau) \otimes C & = & C & \text{if } C(X) = \tau \\ [Q_1] \otimes [Q_2] & = & [Q_1, Q_2] & \text{if } \mathit{DN}(Q_1) \perp \mathit{DN}(Q_2) \\ [?X : \tau, Q_1] \otimes [Q_2] & = & [Q_1] \otimes [Q_2] & \text{if } Q_2(X) = \tau \end{array}$$

It differs from component merging, because it checks that the types of matching specifications are the same.

Other rules Rule T-CLOSE types instantiation. Given a complete mixin module of type $\langle M^c \rangle$, close makes it into a record. The type of the result is $\{\lceil M^c \rceil\}$, which is obtained by flattening the blocks in M^c, forgetting the ! flags.

Rule T-DELETE types deletion. For a mixin module e of type $\langle M \rangle$, the rule gives $e_{|-X}$ the type $\langle \mathit{Del}(M, X) \rangle$, in which $\mathit{Del}(M, X)$ denotes M, where any declaration of the shape $!X : \tau$ is replaced with $?X : \tau$.

The other typing rules are straightforward.

Soundness The type sytem is sound, in the sense that the following lemmas and theorem hold.

Lemma 1 (Subject reduction)
If $\Gamma \vdash e : \tau$ *and* $e \to e'$, *then* $\Gamma \vdash e' : \tau$.

Lemma 2 (Progress)
If $\emptyset \vdash e : \tau$, *then either* e *is a value, or there exists* e' *such that* $e \to e'$.

Theorem 1 (Soundness)
If $\emptyset \vdash e : \tau$, *then either* e *reduces to a value, or its evaluation does not terminate.*

5 Related work

Kernel calculi with mixin modules The idea of mixin modules comes from that of mixins, introduced by Bracha [4] as a model of inheritance. In this model, called Jigsaw, classes are represented by *mixins*, which are equipped with a powerful set of modularity operations, and can be instantiated into *objects*. Mixins are syntactically restricted to contain only values, which makes them as restrictive as classes. What differentiates them from classes is their cleaner design, which gave other authors the idea to generalize them to handle modules as well as objects.

Ancona and Zucca [2] propose a call-by-name module system based on some of Bracha's ideas, called *CMS*. [2] In *CMS* mixin modules, the definitions may be arbitrary expressions. However, the semantics of modules in *CMS* does not model modules in a call-by-value setting. Indeed, the evaluation of call-by-value modules is immediate, whereas in *CMS*, evaluation is triggered by component selection. The problem is that it is not possible to simply state that modules are evaluated as soon as possible in *CMS* to obtain a call-by-value version of it. Indeed, there are cases where evaluating mixin modules contradicts their late-binding semantics. For instance, consider the module $\langle x = 0, y = x + 1 \rangle$. The usual call-by-value semantics of modules evaluates it to $v = \langle x = 0, y = 1 \rangle$, whose components cannot be overridden anymore. For instance, the action of overriding x with 1 should yield the module $\langle x = 1, y = x + 1 \rangle$, which cannot be recovered from v. Thus, the separation we make in *Mix* between mixin modules and modules is necessary, and makes a direct adaptation of *CMS* to call-by-value inappropriate.

This separation can be encoded in Wells and Vestergaard's **m**-calculus [21]. Still, **m** does not restrict recursive definitions at all, and does not allow the user to specify an order of evaluation. *Mix* can be seen as a specialization of **m** with explicit distinction between mixin modules and modules, built-in late binding behavior, restricted recursive definitions, user-specified order of evaluation, and a sound type system.

[2] A monadic version of *CMS* with side effects [1] has been designed more recently. For our purpose, the distinction with *CMS* does not matter.

Language designs with mixin modules Duggan and Sourelis [8] propose an extension of ML with mixin modules, where mixin modules are divided into a *prelude*, a *body*, and an *initialization section*. Only definitions from the body are concerned by mixin module composition, the other sections being simply concatenated (and disjoint). Also, the body is restricted to functions and data-type definitions, which prevents illegal recursive definitions from arising dynamically. This is less flexible than *Mix*, since it considerably limits the interleaving of functional and computational definitions.

Flatt and Felleisen [9] introduce the closely related notion of *units*, in the form of (1) a theoretical extension to Scheme and ML and (2) an actual extension of their PLT Scheme implementation of Scheme [20]. In their theoretical work, they only permit values as unit components, except for a separate initialization section. This is more restrictive than *Mix*, in the same way as Duggan and Sourelis. In the implementation, however, the semantics is different. Any expression is allowed as a definition, and instantiation works in two phases. First, all fields are initialized to `nil`; and second, they are evaluated and updated, one after another. This yields both unexpected behavior (consider the definition `x = cons(1, x)`), and dynamic type errors (consider `x = x + 1`), which do not occur in *Mix*. Finally, units do not feature late binding, contrarily to *Mix*.

Linking calculi Other languages that are close to mixin modules are linking calculi [5, 17]. Generally, they support neither nested modules nor late binding, which significantly departs from *Mix*. Furthermore, among them, Cardelli's proposal does not restrict recursion at all, but the operational semantics is sequential in nature and does not appear to handle cross-unit recursion. As a result, the system seems to lack the progress property. Finally, Machkasova and Turbak [17] explore a linking calculus with a very rich equational theory, but that does not restrict recursion either, is not typed, and does not support nested modules.

Flexible mixin modules In the latest versions of flexible mixin modules [11], a solution to the problem of dependency graphs in types is proposed. Instead of imposing that the graph in a mixin module type exactly reflect the dependencies of the considered mixin module, it is rather seen as a bound on its dependencies, thanks to an adequate notion of subtyping. Roughly, it ensures that the considered mixin module has no more dependencies than the graph exposed by its type. This allows two practical techniques for preventing *MM* mixin module types from being heavy and over-specified. First, the interfaces of a mixin module e can be given more constrained dependency graphs than that of e. This makes interfaces more robust to later changes. Second, a certain class of dependency graphs is characterized, that bear a convenient syntactic description, thus avoiding users to explicitly write graphs by hand. In fact, this syntactic sugar allows to write *MM* types exactly as *Mix* types. We call MM^2 the language obtained by restricting *MM* to such types (in a way that remains to be made precise, for instance by insertion of implicit coercions). Comparing the three languages, we distinguish the following criteria.

(A posteriori reordering) The power of MM^2 w.r.t. reordering lies between *MM* and *Mix*. Intuitively, *Mix* does not feature *a posteriori* reordering

of components, while MM^2 does, to a certain extent. Indeed, the interpretation of *Mix* types as *MM* types forces any component preceding a definition to remain there, but inputs are not concerned by this rule. For instance, the *Mix* structure $e_1 = \langle ?X : int, ?Y : int \rangle$ cannot be composed with $e_2 = \langle !Y \rhd y = 0, !X \rhd x = 0 \rangle$, which is unfortunate. This composition is possible in *MM*, of course, but also in MM^2, since e_1 still has an empty dependency graph.

(Over-specified types) *MM* types are clearly over-specified, as evoked in section 1.2. In contrast, MM^2 and *Mix* types are not required to exactly match the actual dependencies of mixin module components.

(Weight) *MM* types are clearly impractical for a user to write by hand. MM^2 and *Mix* types are equivalently economical w.r.t. standard module types.

(Natural side effects) In *MM* and MM^2, in order to force some order of evaluation between two mixin module components, annotations must be inserted. In *Mix*, it is enough to define the components in the expected order, which is more natural.

(Efficiency) In *MM*, composition is linear, but instantiation is at least quadratic. In *Mix*, instantiation is linear, but composition is $O(n \ log \ n)$, if n is the size of the input.

In summary, *MM* can be ruled out, because of its over-specified, impractically heavy types. Between, MM^2 and *Mix*, the choice is less obvious, in spite of *Mix*'s simplicity, more natural handling of side effects and greater efficiency. Indeed, the flexibility gained by MM^2 might turn out significant in practice.

6 Future work

Type components, polymorphism, subtyping Before, to incorporate mixin modules into a practical language, which is our long-term goal, we have to refine our type system in at least two directions. First, we have to design an extended version of *Mix* including ML style user-defined type components and data types. This task should benefit from recent advances in the design of recursive module systems [6, 7]. Second, we have to enrich our type system with notions of subtyping and polymorphism over mixin modules. Indeed, it might turn out too restrictive for a module system to require, as *Mix* does, a definition filling a declaration of type τ to have exactly type τ.

Compilation The *Mix* language features anonymous definitions, and thus the compilation scheme for mixin modules proposed by Hirschowitz and Leroy [12] does not apply. A possible extension of this scheme to anonymous definitions is sketched in later work [11], but not formalized. This extension might apply to *Mix*. However, it should be possible to do better than this, by taking advantage of the more rigid structure of *Mix* mixin modules.

User-defined composition The composition operator of *Mix* is somewhat arbitrary. This gives the idea to explore the possibility of allowing user-defined composition functions. These could be written as mixin modules importing the proper (meta) types of mixtures, components, ... as well as the appropriate constructors and destructors over these types. Such mixin modules would then be converted into composition operators by a dedicated operator of the language.

References

[1] Davide Ancona, Sonia Fagorzi, Eugenio Moggi, and Elena Zucca. Mixin modules and computational effects. In *International Colloquium on Automata, Languages and Programming*, 2003.

[2] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.

[3] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

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

[5] Luca Cardelli. Program fragments, linking, and modularization. In *24th symposium Principles of Programming Languages*, pages 266–277. ACM Press, 1997.

[6] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Programming Language Design and Implementation*, pages 50–63. ACM Press, 1999.

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

[8] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *International Conference on Functional Programming*, pages 262–273. ACM Press, 1996.

[9] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

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

[11] Tom Hirschowitz. Call-by-value mixin modules (preliminary version). Available on the Web, `http://pauillac.inria.fr/~hirschow/phd/state.html`, 2003.

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

[13] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *International Conference on Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

[14] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. A reduction semantics for call-by-value mixin modules. Research report RR-4682, INRIA, January 2003.

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

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

[17] Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2000.

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

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

[20] The PLT. PLT Scheme. Available on the Web, `http://www.plt-scheme.org/`.

[21] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 412–428. Springer-Verlag, 2000.

A Preliminary definitions

We extend the functions Add and $\otimes$ to enriched signatures and specifications.

$$
\begin{array}{rcl}
Add(\varepsilon, {M^e}_1, {M^e}_2) & = & {M^e}_1, {M^e}_2 \\
Add((M^e_1, C^e), M^e_2, M^e_3) & = & Add(M^e_1, M^e_2, (C^e, M^e_3)) \\
 & & \quad \text{if } DN(C^e) \perp DN(M^e_2, M^e_3) \\
Add(\,(M^e_1, C^e_1), & = & Add(\,M^e_1, \\
\quad (M^{e1}_2, C^e_2, M^{e2}_2), & & \quad M^{e1}_2, \\
\quad M^e_3) & & \quad (C^e_1 \otimes C^e_2, M^{e2}_2, M^e_3)) \\
\multicolumn{3}{c}{\text{if } DN(C^e_1) \perp DN(M^{e1}_2, M^{e2}_2, M^e_3) \text{ and } DN(C^e_1) \cap DN(C^e_2) \neq \emptyset}
\end{array}
$$

$$
\begin{array}{rcll}
{C^e}_1 \otimes {C^e}_2 & = & {C^e}_2 \otimes {C^e}_1 & \\
(?X \rhd x : \tau) \otimes C^e & = & C^e & \text{if } C^e(X \rhd x) = \tau \\
[{Q^e}_1] \otimes [{Q^e}_2] & = & [{Q^e}_1, {Q^e}_2] & \text{if } DN({Q^e}_1) \perp DN({Q^e}_2) \\
[?X \rhd x : \tau, {Q^e}_1] \otimes [{Q^e}_2] & = & [{Q^e}_1] \otimes [{Q^e}_2] & \text{if } {Q^e}_2(X \rhd x) = \tau
\end{array}
$$

B Stability

Proposition 2 (Stability) *If e is a term and $e \to e'$, then e' is a term.*

Proof The only non-trivial case is rule COMPOSE.

It is enough to show that if $DN(m_3) \perp DN(m_1) \cup DN(m_2)$ and m_1, m_2, and m_3 are correct and pairwise compatible, in the sense of section 3.2, then $Add(m_1, m_2, m_3)$ is correct, if defined.

We do it by induction on m_1, assuming that $Add(m_1, m_2, m_3)$ is defined.

- The base case, $m_1 = \varepsilon$, is trivial.
- If $m_1 = (m_1^0, c)$ with $DN(c) \perp DN(m_2)$, then m_1^0, m_2, and (c, m_3) are correct mixtures (because $DN(c) \perp DN(m_3)$ and $m_1 \Cdot m_3$, so $DV(c) \perp DV(m_3)$). Moreover, we have $DN(c, m_3) \perp DN(m_1^0) \cup DN(m_2)$, since m_1 is correct. So, by induction hypothesis, $Add(m_1^0, m_2, (c, m_3))$ is correct. As it is equal by definition to $Add(m_1, m_2, m_3)$, the latter is also correct.
- If $m_1 = (m_1^0, c_1)$, $m_2 = (m_2^0, c_2, m_2^1)$, $DN(c_1) \cap DN(c_2) \neq \emptyset$, and $DN(c_1) \perp DN(m_2^0)$, then m_1^0, m_2^0, and $(c_1 \otimes c_2, m_2^1, m_3)$ are correct mixtures, since
  - $DN(c_1 \otimes c_2) = DN(c_1) \cup DN(c_2) \perp DN(m_3)$,
  - $DN(m_2^1) \perp DN(m_3)$,
  - $DN(c_2) \perp DN(m_2^1)$,
  - and $DN(c_1) \perp DN(m_2^1)$ since $Add(m_1, m_2, m_3)$ is defined.

End proof

C Subject reduction

Lemma 3 (Weakening)

- *If $\Gamma \vdash e : \tau$ and $FV(e) \perp dom(\Gamma')$, then $\Gamma + \Gamma' \vdash e : \tau$.*
- *If $\Gamma \vdash c : C^e$ and $FV(c) \cup DV(c) \perp dom(\Gamma')$, then $\Gamma + \Gamma' \vdash c : C^e$.*
- *If $\Gamma \vdash m : M^e$ and $FV(m) \cup DV(m) \perp dom(\Gamma')$, then $\Gamma + \Gamma' \vdash m : M^e$.*
- *If $\Gamma \vdash b : \Gamma_b$ and $FV(b) \cup DV(b) \perp dom(\Gamma')$, then $\Gamma + \Gamma' \vdash b : \Gamma_b$.*

Lemma 4 (Substitution)

- *If $\Gamma + \{x \mapsto \tau_x\} \vdash e : \tau$ (1) and $\Gamma \vdash v : \tau_x$ (2), then $\Gamma \vdash \{x \mapsto v\}(e) : \tau$.*
- *If $\Gamma + \{x \mapsto \tau_x\} \vdash c : C^e$ (1), $x \notin DV(c)$, and $\Gamma \vdash v : \tau_x$ (2), then $\Gamma \vdash \{x \mapsto v\}(c) : C^e$.*

- *If* $\Gamma + \{x \mapsto \tau_x\} \vdash m : M^e$ *(1), $x \notin DV(m)$, and* $\Gamma \vdash v : \tau_x$ *(2), then* $\Gamma \vdash \{x \mapsto v\}(m) : M^e$.
- *If* $\Gamma + \{x \mapsto \tau_x\} \vdash b : \Gamma_b$ *(1) $x \notin DV(b)$, and* $\Gamma \vdash v : \tau_x$ *(2), then* $\Gamma \vdash \{x \mapsto v\}(b) : \Gamma_b$.

Proof By mutual induction on the typing derivation. The base cases and almost all the proofs for outputs, bindings and components are easy, so we omit them.

*(*T-Record*)* Assume $e = \{s\}$. By induction hypothesis, for all $X \in dom(s)$, we have $\Gamma \vdash \{x \mapsto v\}(s(X)) : \tau$ and $dom(s) = dom(S)$, so we derive $\Gamma \vdash \{x \mapsto v\}(\{s\}) : \tau$.

*(*T-Struct,T-Compose,T-Close,T-Select*)* Work similarly by induction hypothesis.

*(*T-LetRec*)* Assume $e = \mathsf{letrec}\ b\ \mathsf{in}\ e_1$. By induction hypothesis, we obtain $\Gamma + \Gamma_b \vdash \{x \mapsto v\}(b) : \Gamma_b$ and $\Gamma + \Gamma_b \vdash \{x \mapsto v\}(e_1) : \Gamma_b$ (where the domain of b has been α-converted to fresh variables). We have $\{x \mapsto v\}(e) = \mathsf{letrec}\ \{x \mapsto v\}(b)\ \mathsf{in}\ \{x \mapsto v\}(e_1)$, which gives the expected result.

*(*T-Let*)* Assume $e = (\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)$. By induction hypothesis, $\Gamma \vdash \{x \mapsto v\}(e_1) : \tau_1$ and $\Gamma + \{y \mapsto \tau_1\} \vdash e_2 : \tau$, which immediately gives the expected result.

*(*T-Block*)* Assume $m = ([q], m_1)$. By induction hypothesis, we have $\Gamma + \Gamma_q \vdash \{x \mapsto v\}(m) : M^e$ and for all $u \in q$, $\Gamma + \Gamma_q \vdash \{x \mapsto v\}(u) : {U^e}_u$. Moreover, as valid recursive expressions are assumed to be stable by substitution, we have for all $u \in q$, $\{x \mapsto v\}(u) \in RecExp_?$, which allows to derive $\Gamma \vdash \{x \mapsto v\}(m) : M^e$.

End proof

Lemma 5 (Parallel substitution)
If

$$\begin{array}{ll} \Gamma + \Gamma' \vdash e : \tau & (1) \\ \textit{For all } x \in dom(\Gamma'),\ \Gamma \vdash e_x : \tau_x & (2) \\ \textit{For all } x \neq y \in dom(\Gamma'),\ x \notin FV(e_y) & (3) \end{array}$$

then

$$\Gamma \vdash \{x \mapsto e_x \mid x \in dom(\Gamma')\}(e) : \tau$$

Proof Let $\Gamma' = \{x_1 \mapsto \tau_1\} + \ldots + \{x_n \mapsto \tau_n\}$, for $i \in \{1 \ldots n\}$, $e_i = e_{x_i}$, and $\sigma = \{x_i \mapsto e_i \mid i \in \{1 \ldots n\}\}$. By hypothesis (3), we have $\sigma = \sigma_1 \circ \ldots \circ \sigma_n$, where for $i \in \{1 \ldots n\}$, $\sigma_i = \{x_i \mapsto e_i\}$. Thus, by trivial induction on n and lemma 4, we obtain the desired result. **End proof**

Proposition 3
If $\Gamma \vdash m_1 : {M^e}_1$, $\Gamma + Env({M^e}_1) \vdash m_2 : {M^e}_2$, *then* $\Gamma \vdash (m_1, m_2) : ({M^e}_1, {M^e}_2)$.

In the following, we implicitly extend Add and $\otimes$ to compatible enriched signatures, in the straightforward way.

Proposition 4
If both are defined, then $Add(M^e{}_1, M^e{}_2, M^e{}_3) = (Add(M^e{}_1, M^e{}_2, \varepsilon), M^e{}_3)$.

Proposition 5
If $\Gamma \vdash (m_1, c) : (M^e{}_1, C^e)$, *then* $\Gamma \vdash m_1 : M^e{}_1$ *and* $\Gamma + Env(M^e{}_1) \vdash c : C^e$.

Proposition 6
If $\Gamma \vdash (m_1, c, m_2) : M^e$, *then there exist* $M^e{}_1, C^e$, *and* $M^e{}_2$ *such that* $\Gamma \vdash m_1 : M^e{}_1$, $\Gamma + Env(M^e{}_1) \vdash c : C^e$, $\Gamma + Env(M^e{}_1, C^e) \vdash m_2 : M^e{}_2$.

Proposition 7
For all $M^e{}_1, M^e{}_2, M^e{}_3$, *if* $M^e = Add(M^e{}_1, M^e{}_2, M^e{}_3)$, *then for any* $i = 1, 2, 3$, $Env(M^e{}_i) \subseteq Env(M^e)$.

Lemma 6 (Merge)
If $\Gamma \vdash c_1 : C^e{}_1$
$\Gamma \vdash c_2 : C^e{}_2$
$c_1 \otimes c_2 = c$
$C^e{}_1 \otimes C^e{}_2 = C^e$
and $c_1 \Rsh c_2$,
then $\Gamma \vdash c : C^e$

Proof By induction on the proof of $c_1 \otimes c_2 = c$.

- If the proof ends with an application of the commutativity rule, then, as $\otimes$ is also commutative on enriched specifications, we obtain the desired result by induction hypothesis.
- Assume $c_1 = (X \rhd x = \bullet)$ and $dom(c_2)(X) = x$. By the typing rules, we get $C^e{}_1 = (?X \rhd x : \tau_x)$, and by definition of *Add*, $C^e{}_2(X \rhd x) = \tau_x$, and $C^e = C^e{}_2$. Moreover, by definition of $\otimes$, $c = c_2$, so we obtain $\Gamma \vdash c : C^e$ directly.
- Assume $c_1 = [q_1]$, $c_2 = [q_2]$, and $DN(q_1) \perp DN(q_2)$. We have $c = [q_1, q_2]$. Moreover, by the typing rules, there exists $Q^e{}_1$ and $Q^e{}_2$ such that $C^e{}_1 = [Q^e{}_1]$ and $C^e{}_2 = [Q^e{}_2]$. By definition of $\otimes$, we obtain $C^e = [Q^e{}_1, Q^e{}_2]$. Let $Q^e = (Q^e{}_1, Q^e{}_2) = \biguplus_{u \in (q_1, q_2)} U^e{}_u$. By typing, for all $u \in (q_1, q_2)$, $u \in RecExp_?$ and $\Gamma \vdash u : U^e{}_u$. Thus, we can derive $\Gamma \vdash c : C^e$.
- Assume $c_1 = [X \rhd x = \bullet, q_1]$, $c_2 = [q_2]$, and $dom(q_2)(X) = x$. By typing, $C^e{}_1 = [?X \rhd x : \tau_x, Q^e{}_1]$ and $C^e{}_2 = [Q^e{}_2]$, so $C^e{}_1 \otimes C^e{}_2 = [Q^e{}_1] \otimes [Q^e{}_2]$ and $Q^e{}_2(X \rhd x) = \tau_x$. By induction hypothesis, we derive $\Gamma \vdash [q_1] \otimes [q_2] : C^e$, which gives the desired result.

End proof

Lemma 7 (Add)
If

$$\Gamma \vdash m_1 : M^e{}_1 \quad (1)$$
$$\Gamma \vdash m_2 : M^e{}_2 \quad (2)$$
$$\Gamma + Env(M^e) \vdash m_3 : M^e{}_3 \quad (3)$$
$$Add(m_1, m_2, m_3) \text{ is defined} \quad (4)$$

with $M^e = Add(M^e{}_1, M^e{}_2, \varepsilon)$
$m_1 \Rightarrow m_2$
$DN(m_3) \perp DN(m_1, m_2)$
$DV(m_3) \perp DV(m_1, m_2)$
then $\Gamma \vdash Add(m_1, m_2, m_3) : Add(M^e{}_1, M^e{}_2, M^e{}_3)$.

Proof By induction on m_1.

(Base) Assume $m_1 = \varepsilon$. Then $M^e = M^e{}_2$, $Add(m_1, m_2, m_3) = (m_2, m_3)$, and $Add(M^e{}_1, M^e{}_2, M^e{}_3) = (M^e{}_2, M^e{}_3)$, so proposition 3 gives the expected result.

(Induction) Assume $m_1 = (m_1^0, c)$.

- If $DN(c) \perp DN(m_2)$, since we also know $DN(c) \perp DN(m_3)$ by hypothesis, we obtain by definition of Add that $m = Add(m_1, m_2, m_3) = Add(m_1^0, m_2, (c, m_3))$. But by proposition 5 and (1), $M^e{}_1 = (M^{e\,0}{}_1, C^e)$, $\Gamma + Env(M^{e\,0}{}_1) \vdash c : C^e$, and $\Gamma \vdash m_1^0 : M^{e\,0}{}_1$.
  Thus, by lemma 3 and proposition 7, we obtain

  $$\Gamma + Env(Add(M^{e\,0}{}_1, M^e{}_2, \varepsilon)) \vdash c : C^e.$$

  But by proposition 4, $M^e = Add(M^e{}_1, M^e{}_2, \varepsilon) = Add(M^{e\,0}{}_1, M^e{}_2, C^e) = (Add(M^{e\,0}{}_1, M^e{}_2, \varepsilon), C^e)$, so (3) is equivalent to

  $$\Gamma + Env(Add(M^{e\,0}{}_1, M^e{}_2, \varepsilon)) + Env(C^e) \vdash m_3 : M^e{}_3,$$

  and so

  $$\Gamma + Env(Add(M^{e\,0}{}_1, M^e{}_2, \varepsilon)) \vdash (c, m_3) : (C^e, M^e{}_3).$$

  So, by induction hypothesis we obtain

  $$\Gamma \vdash Add(m_1^0, m_2, (c, m_3)) : Add(M^{e\,0}{}_1, M^e{}_2, (C^e, M^e{}_3)),$$

  which is equivalent to the desired result.
- Otherwise, $m_2 = (m_2^0, c', m_2^1)$ and $Add(m_1, m_2, m_3) = Add(m_1^0, m_2^0, (c \otimes c', m_2^1, m_3))$, with $DN(c) \perp DN(m_2^0, m_2^1)$ and $DN(c) \cap DN(c') \neq \emptyset$.
  By proposition 6, $M^e{}_1 = (M^{e\,0}{}_1, C^e)$ and $M^e{}_2 = (M^{e\,0}{}_2, C^{e\prime}, M^{e\,1}{}_2)$, such that
  $\Gamma \vdash m_1^0 : M^{e\,0}{}_1$, $\Gamma + Env(M^{e\,0}{}_1) \vdash c : C^e$,
  $\Gamma \vdash m_2^0 : M^{e\,0}{}_2$, $\Gamma + Env(M^{e\,0}{}_2) \vdash c' : C^{e\prime}$, and
  $\Gamma + Env(M^{e\,0}{}_2, C^{e\prime}) \vdash m_2^1 : M^{e\,1}{}_2$.
  But by lemma 3, letting $M^{e\,0} = Add(M^{e\,0}{}_1, M^{e\,0}{}_2, \varepsilon)$ and $C^{e\prime\prime} = C^e \otimes C^{e\prime}$, we have
  $M^e = (M^{e\,0}, C^{e\prime\prime}, M^{e\,1}{}_2)$,
  $\Gamma + Env(M^{e\,0}) \vdash c : C^e$,
  $\Gamma + Env(M^{e\,0}) \vdash c' : C^{e\prime}$,
  $\Gamma + Env(M^{e\,0}, C^{e\prime\prime}) \vdash m_2^1 : M^{e\,1}{}_2$, and
  $\Gamma + Env(M^{e\,0}, C^{e\prime\prime}, M^{e\,1}{}_2) \vdash m_3 : M^e{}_3$.

By lemma 6, we have $\Gamma + Env(M^{e}{}^{0}) \vdash c \otimes c' : C^{e\prime\prime}$, so by induction hypothesis we obtain

$$\Gamma \vdash Add(m_1^0, m_2^0, (c \otimes c', m_2^1, m_3)) : Add(M^{e}{}_1^0, M^{e}{}_2^0, (C^{e\prime\prime}, M^{e}{}_2^1, M^{e}{}_3)),$$

which is equivalent to the desired result.

End proof

Lemma 8 (Close)
If $\Gamma \vdash m^c : M^e$, *then* $\Gamma \vdash Bind(m^c, \{\lceil m^c \rceil\}) : \{\lceil M^e \rceil\}$.

Proof Let $[\![M^e]\!]$ be the natural record sequence that can be extracted from an enriched signature M^e, namely, the one such that $[\![\delta X \rhd x : \tau]\!] = (X = x)$ and $[\![\delta\_ \rhd x : \tau]\!] = \varepsilon$.

We prove the more general property that given some complete enriched block Q^e, if

$$\Gamma + Env(Q^e) \vdash m^c : M^e \quad (1)$$
$$DN(m^c) \perp DN(Q^e) \quad (2)$$
$$DV(m^c) \perp DV(Q^e) \quad (3)$$

then

$$\Gamma + Env(Q^e) \vdash Bind(m^c, \{m^c + [\![Q^e]\!]\}) : \{\lceil M^e \rceil + \lceil Q^e \rceil\}.$$

We proceed by induction on m^c. Let $\Gamma' = Env(Q^e)$.

(Base) Assume $m^c = \varepsilon$. Then we just want to prove $\Gamma + \Gamma' \vdash \{[\![Q^e]\!]\} : \{\lceil Q^e \rceil\}$. By rule T-Record, the following two checks are enough

- $dom([\![Q^e]\!]) = dom(\lceil Q^e \rceil)$, which is trivial.
- For all $X \in dom(\lceil Q^e \rceil)$, $\Gamma + \Gamma' \vdash [\![Q^e]\!](X) : \lceil Q^e \rceil(X)$, which follows from the definitions of $[\![\cdot]\!]$ and $\lceil\cdot\rceil$.

(Induction, single) Assume $m^c = (u, m^c{}_1)$. Hypothesis (1) becomes $\Gamma + \Gamma' \vdash (u, m^c{}_1) : M^e$, which gives by inversion

$$M^e = (Sig(U^e), M^e{}_1), \quad (1.1)$$
$$\Gamma + \Gamma' \vdash u : U^e, \quad (1.2)$$
$$\Gamma + \Gamma' + Env(U^e) \vdash m^c{}_1 : M^e{}_1. \quad (1.3)$$

Let $Q^e{}_1 = Q^e, U^e$. We have $\Gamma + Env(Q^e{}_1) \vdash m^c{}_1 : M^e{}_1$, so by induction hypothesis

$$\Gamma + Env(Q^e{}_1) \vdash Bind(m^c{}_1, \{\lceil m^c{}_1 \rceil + [\![Q^e{}_1]\!]\}) : \{\lceil Q^e{}_1 \rceil + \lceil M^e{}_1 \rceil\}.$$

But $\lceil m^c{}_1 \rceil + [\![Q^e{}_1]\!] = \lceil m^c{}_1 \rceil + [\![Q^e]\!] + [\![U^e]\!] = \{m^c\} + [\![Q^e]\!]$,
and $\lceil Q^e{}_1 \rceil + \lceil M^e{}_1 \rceil = \lceil Q^e \rceil + \lceil M^e \rceil$.

Now, as m^c is complete, we know that u is a definition $u = (L \rhd x = e)$. Then $U^e = (!L \rhd x : \tau_x)$ by (1.2), and $\Gamma + \Gamma' \vdash e : \tau_x$. But here, $Env(U^e) = \{x \mapsto \tau_x\}$, so we can derive

$$\frac{\Gamma + \Gamma' \vdash e : \tau_x \qquad \Gamma + \Gamma' + \{x \mapsto \tau_x\} \vdash Bind(m^c{}_1, \{\lceil m^c \rceil + [\![Q^e]\!]\}) : \{\lceil Q^e \rceil + \lceil M^e \rceil\}}{\Gamma + \Gamma' \vdash \mathsf{let}\ x = e\ \mathsf{in}\ Bind(m^c{}_1, \{\lceil m^c \rceil + [\![Q^e]\!]\}) : \{\lceil Q^e \rceil + \lceil M^e \rceil\}}$$

which is equivalent to the expected result.

(Induction, block) Assume $m^c = ([q], m^c{}_1)$. Then, $Bind(m^c, \{\lceil m^c \rceil + [\![Q^e]\!]\}) = \mathsf{letrec}\ \lfloor q \rfloor\ \mathsf{in}\ Bind(m^c{}_1, \{\lceil m^c \rceil + [\![Q^e]\!]\})$.

By inversion, we get $\{U^e{}_u \mid u \in q\}$ such that with $\Gamma'' = \Gamma + \Gamma' + \Gamma_q$, $Q^e{}_q = \biguplus_{u \in q} U^e{}_u$, $M^e = ([Sig(Q^e{}_q)], M^e{}_1)$, and $\Gamma_q = Env(Q^e{}_q)$, the following holds:

$$\Gamma'' \vdash m^c{}_1 : M^e{}_1 \tag{1.1}$$
$$\text{For all } u \in q, \quad u \in RecExp_? \tag{1.2}$$
$$\text{and } \Gamma'' \vdash u : U^e{}_u \tag{1.3}$$

Let $Q^e{}_1 = (Q^e, Q^e{}_q)$. We have $\Gamma'' = \Gamma + Env(Q^e{}_1)$. By induction hypothesis, $\Gamma'' \vdash Bind(m^c{}_1, \{\lceil m^c{}_1 \rceil + [\![Q^e{}_1]\!]\}) : \{\lceil M^e{}_1 \rceil + \lceil Q^e{}_1 \rceil\}$.

But we have $\lceil m^c{}_1 \rceil + [\![Q^e{}_1]\!] = \lceil m^c{}_1 \rceil + [\![Q^e]\!] + [\![Q^e{}_q]\!]$, and $[\![Q^e{}_q]\!] = \lceil [q] \rceil$, so $\lceil m^c{}_1 \rceil + [\![Q^e{}_1]\!] = \lceil m^c \rceil + [\![Q^e]\!]$.

Similarly, $\lceil M^e{}_1 \rceil + \lceil Q^e{}_1 \rceil = \lceil M^e{}_1 \rceil + \lceil Q^e \rceil + \lceil Q^e{}_q \rceil = \lceil M^e \rceil + \lceil Q^e \rceil$, so, by rule T-LetRec and by definition of *Bind*, it is enough to derive $\Gamma'' \vdash \lfloor q \rfloor : \Gamma_q$, which follows from (1.3).

End proof

Lemma 1 (Subject reduction) *If $\Gamma \vdash e_1 : \tau$ (1) and $e_1 \to e_2$ (2), then $\Gamma \vdash e_2 : \tau$.*

Proof By induction on the proof of $e_1 \to e_2$, and case analysis on the last rule of the derivation.

*(*Context*)* Assume $e_1 = \mathbb{E}[e_3]$ and $e_2 = \mathbb{E}[e_4]$, with $e_3 \to e_4$. We proceed by case analysis on $\mathbb{E}$.

- Assume $\mathbb{E} = \{s^v, X = \square, s\}$. Then by inversion of (1), we have

$$dom(s^v, X = \square, s) = dom(S) \tag{1.1}$$
$$\text{For all } Y \in dom(s^v, s), \Gamma \vdash (s^v, s)(Y) : S(Y) \tag{1.2}$$
$$\Gamma \vdash e_3 : S(X) \tag{1.3}$$

  By induction hypothesis, we obtain that $\Gamma \vdash e_4 : S(X)$, so we derive $\Gamma \vdash e_2 : \tau$.

- The case $\mathbb{E} = \Box.X$ similarly works by induction hypothesis.
- Assume $\mathbb{E} = (\mathsf{let}\ x = \Box\ \mathsf{in}\ e)$. By inversion of (1), we obtain $\Gamma \vdash e_3 : \tau_3$ and $\Gamma + \{x \mapsto \tau_3\} \vdash e : \tau$, and by induction hypothesis, we get $\Gamma \vdash e_4 : \tau_3$, so we derive $\Gamma \vdash e_2 : \tau$.
- Assume $\mathbb{E} = \Box \gg e$. By inversion of (1), we obtain $\Gamma \vdash e_3 : \langle M_1 \rangle$ and $\Gamma \vdash e : \langle M_2 \rangle$, with $\tau = \langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle$. Thus, by induction hypothesis, we get $\Gamma \vdash e_4 : \langle M_1 \rangle$, so we derive the desired result.
- Assume $\mathbb{E} = \mathsf{close}\ \Box$. Then, by inversion of (1), we obtain $\Gamma \vdash e_3 : \langle M^c \rangle$, with $\tau = \{\lceil M^c \rceil\}$. But by induction hypothesis, we get $\Gamma \vdash e_4 : \langle M^c \rangle$, so we derive the expected result.

*(*Close*)* Assume $e_1 = \mathsf{close}\ \langle m^c \rangle$ and $e_2 = \mathit{Bind}(m^c, \{\lceil m^c \rceil\})$. By inversion of (1), we get

$$\Gamma \vdash \langle m^c \rangle : \langle M^c \rangle \tag{1.1}$$
$$\tau = \{\lceil M^c \rceil\} \tag{1.2}$$

By inversion of (1.1), we obtain $\Gamma \vdash m^c : M^e$, for some M^e such that $M^c = \mathit{Sig}(M^e)$. By lemma 8, we can derive $\Gamma \vdash \mathit{Bind}(m^c, \{\lceil m^c \rceil\}) : \{\lceil M^c \rceil\}$, which is in fact $\Gamma \vdash e_2 : \tau$.

*(*Let*)* Assume $e_1 = (\mathsf{let}\ x = v\ \mathsf{in}\ e)$ an $e_2 = \{x \mapsto v\}(e)$. By inversion of (1), we obtain

$$\Gamma \vdash v : \tau_v \tag{1.1}$$
$$\Gamma + \{x \mapsto \tau_v\} \vdash e : \tau \tag{1.2}$$

By lemma 4, we derive $\Gamma \vdash \{x \mapsto v\}(e) : \tau$.

*(*LetRec*)* Assume $e_1 = \mathsf{letrec}\ b\ \mathsf{in}\ e$ and $e_2 = \{x \mapsto \mathsf{letrec}\ b\ \mathsf{in}\ b(x) \mid x \in \mathit{dom}(b)\}(e)$. By inversion of (1), we obtain Γ_b such that

$$\Gamma + \Gamma_b \vdash b : \Gamma_b \tag{1.1}$$
$$\Gamma + \Gamma_b \vdash e : \tau \tag{1.2}$$

By inversion of (1.1), we also obtain that for any $x \in \mathit{dom}(b)$,

$$\Gamma + \Gamma_b \vdash b(x) : \Gamma_b(x)$$

and therefore we can derive by rule T-LetRec

$$\Gamma \vdash \mathsf{letrec}\ b\ \mathsf{in}\ b(x) : \Gamma_b(x).$$

Thus, letting $\sigma = \{x \mapsto \mathsf{letrec}\ b\ \mathsf{in}\ b(x) \mid x \in \mathit{dom}(b)\}$, lemma 5 gives

$$\Gamma \vdash \sigma(e) : \tau$$

which is the expected result.

*(*Compose*)* Assume $e_1 = \langle m_1 \rangle \gg \langle m_2 \rangle$ and $e_2 = \langle \mathit{Add}(m_1, m_2, \varepsilon) \rangle$. By inversion of (1), we obtain a typing derivation like

$$\frac{\Gamma \vdash m_1 : {M^e}_1 \qquad \Gamma \vdash m_2 : {M^e}_2}{\dfrac{\Gamma \vdash \langle m_1 \rangle : \langle M_1 \rangle \qquad \Gamma \vdash \langle m_2 \rangle : \langle M_2 \rangle}{\Gamma \vdash \langle m_1 \rangle \gg \langle m_2 \rangle : \langle Add(M_1, M_2, \varepsilon) \rangle}}$$

where $M_1 = Sig({M^e}_1)$
$M_2 = Sig({M^e}_2)$
$\tau = \langle Add(M_1, M_2, \varepsilon) \rangle$.

By lemma 7, we derive

$$\Gamma \vdash Add(m_1, m_2, \varepsilon) : Add({M^e}_1, {M^e}_2, \varepsilon),$$

which gives by rule T-STRUCT

$$\Gamma \vdash \langle Add(m_1, m_2, \varepsilon) \rangle : \langle Sig(Add({M^e}_1, {M^e}_2, \varepsilon)) \rangle.$$

But obviously, we have $Sig(Add({M^e}_1, {M^e}_2, \varepsilon)) = Add(M_1, M_2, \varepsilon)$, which gives the expected result.

*(*DELETE*)* The reduction rule replaces a definition $X \rhd x = e$, typed $!X \rhd x : \tau_x$ for some τ_x, with the declaration $X \rhd x = \bullet$, which can be given the type $?X \rhd x : \tau_x$. This gives exactly the desired type to the reduct.

*(*SELECT*)* This case is trivial.

End proof

D Progress

Lemma 9 (Progress/Merge)
If

$$\begin{array}{ll} \Gamma_1 \vdash c_1 : {C^e}_1 & (1) \\ \Gamma_2 \vdash c_2 : {C^e}_2 & (2) \\ {C^e}_1 \otimes {C^e}_2 \textit{ is defined} & (3) \\ \textit{and } c_1 \Rightarrow c_2 & (4) \end{array}$$

then $c_1 \otimes c_2$ is defined.

Proof Let $C^e = {C^e}_1 \otimes {C^e}_2$. We proceed by induction on the proof of $C^e = {C^e}_1 \otimes {C^e}_2$.

- If the last rule used is commutation, then by induction hypothesis we directly obtain the desired result.
- If the last rule used is the second one, then ${C^e}_1 = (?X \rhd x : \tau)$ and ${C^e}_2(X \rhd x) = \tau$. By inversion of (1) and (2), we know that $c_1 = (X \rhd x = \bullet)$ and $X \in DN(c_2)$, and $dom(c_2)(X) = x$, so $c_1 \otimes c_2$ is defined and is equal to c_2.

- If the last rule used is the third one, then ${C^e}_1 = [{Q^e}_1]$ and ${C^e}_2 = [{Q^e}_2]$, with $DN({Q^e}_1) \perp DN({Q^e}_2)$. But by inversion of (1) and (2), this implies $DN(c_1) \perp DN(c_2)$, so we can apply the corresponding rule at the level of blocks, which ensures that $c_1 \otimes c_2$ is defined.

- If the last rule used is the fourth one, then ${C^e}_1 = [?X \rhd x : \tau, {Q^e}_1]$ and ${C^e}_2 = [{Q^e}_2]$, with ${Q^e}_2(X \rhd x) = \tau$. By inversion of (1) and (2), we obtain $c_1 = [X \rhd x = \bullet, q_1]$ and $c_2 = [q_2]$, with $dom(q_2)(X) = x$.

  Moreover, we have $\Gamma + \{x \mapsto \tau\} \vdash [q_1] : [{Q^e}_1]$, so by induction hypothesis, we obtain that $[q_1] \otimes [q_2]$ is defined.

  Thus, we can apply the fourth rule at the level of blocks to obtain the expected result.

End proof

Lemma 10 (Progress/Compose)
If

$$\begin{array}{l}\Gamma \vdash m_1 : {M^e}_1\\ \Gamma \vdash m_2 : {M^e}_2\\ Add({M^e}_1, {M^e}_2, \varepsilon) \textit{ is defined}\\ \textit{and } m_1 \asymp m_2\end{array}$$

then $Add(m_1, m_2, \varepsilon)$ *is also defined.*

Proof We prove the more general property that if

$$\begin{array}{ll}\Gamma \vdash m_1 : {M^e}_1 & (1)\\ \Gamma \vdash m_2 : {M^e}_2 & (2)\\ DN(m_3) = DN({M^e}_3) & (3)\\ Add({M^e}_1, {M^e}_2, {M^e}_3) \text{ is defined} & (4)\\ \text{and } m_1 \asymp m_2 & (5)\end{array}$$

then $Add(m_1, m_2, m_3)$ is also defined.

We proceed by induction on m_1.

(Base) Assume $m_1 = \varepsilon$. Then, trivially $Add(m_1, m_2, m_3)$ is defined.

(Induction) We distinguish two cases.

- Assume $m_1 = (m_1^0, c)$, with $DN(c) \perp DN(m_2)$. By proposition 5, this gives ${M^e}_1^0$ and C^e such that $\Gamma \vdash m_1^0 : {M^e}_1^0$ and $\Gamma + Env({M^e}_1^0) \vdash c : C^e$, with ${M^e}_1 = ({M^e}_1^0, C^e)$. So, since $Add({M^e}_1, {M^e}_2, {M^e}_3)$ is defined and $DN(c) \perp DN(m_2)$, it must be equal to $Add({M^e}_1^0, {M^e}_2, (C^e, {M^e}_3))$, which implies $DN(C^e) \perp DN({M^e}_3)$. Thus, $DN(c) = DN(C^e) \perp DN({M^e}_3) = DN(m_3)$. Moreover, $DN(c, m_3) = DN(C^e, M_3)$, so, by induction hypothesis, $Add(m_1^0, m_2, (c, m_3))$ is defined. So, $Add(m_1, m_2, m_3)$ is also defined, and equal to the former.

- Assume $m_1 = (m_1^0, c_1)$, with $DN(c_1) \cap DN(m_2) \neq \emptyset$. Then, we can decompose m_2 as (m_2^0, c_2, m_2^1), with $DN(c_1) \perp DN(m_2^0)$ and $DN(c_1) \cap$

$DN(c_2) \neq \emptyset$. By proposition 6, there exist ${M^e}_1^0, {C^e}_1, {M^e}_2^0, {C^e}_2$, and ${M^e}_2^1$ such that

$$\Gamma \vdash m_1^0 : {M^e}_1^0 \quad (1.1)$$
$$\Gamma + Env({M^e}_1^0) \vdash c_1 : {C^e}_1 \quad (1.2)$$
$$\Gamma \vdash m_2^0 : {M^e}_2^0 \quad (2.1)$$
$$\Gamma + Env({M^e}_2^0) \vdash c_2 : {C^e}_2 \quad (2.2)$$
$$\Gamma + Env({M^e}_2^0, {C^e}_2) \vdash m_2^1 : {M^e}_2^1 \quad (2.3)$$

Moreover, since $Add({M^e}_1, {M^e}_2, {M^e}_3)$ is defined, it must be equal to $Add({M^e}_1^0, {M^e}_2^0, ({C^e}_1 \otimes {C^e}_2, {M^e}_2^1, {M^e}_3))$, which is thus defined. Further, by lemma 9, we know that $c_1 \otimes c_2$ is also defined.

Also, we have

$$\begin{aligned} &DN(c_1 \otimes c_2, m_2^1, m_3) \\ &= DN(c_1) \cup DN(c_2) \cup DN(m_2^1) \cup DN(m_3) \\ &= DN({C^e}_1) \cup DN({C^e}_2) \cup DN({M^e}_2^1) \cup DN({M^e}_3) \\ &= DN({C^e}_1 \otimes {C^e}_2, {M^e}_2^1, M_3) \end{aligned}$$

So, by induction hypothesis, $Add(m_1^0, m_2^0, (c_1 \otimes c_2, m_2^1, m_3))$ is defined.

Finally, since $DN(c_1) \perp DN(m_2^1)$ and $DN({C^e}_1) \perp DN({M^e}_2^1)$, we obtain that $Add(m_1, m_2, m_3)$ is defined and equal to $Add(m_1^0, m_2^0, (c_1 \otimes c_2, m_2^1, m_3))$.

End proof

Lemma 2 (Progress) *If $\emptyset \vdash e : \tau$, then either e is a value (a), or $e \to e'$ (b).*

Proof We proceed by induction on the derivation $\emptyset \vdash e : \tau$. Then, we reason by case analysis on the last typing rule of the derivation. The proof relies on the fact that evaluation contexts do not bind any variable.

*(*T-VAR*)* Impossible.

*(*T-RECORD*)* If e is a value, then we are in case (a). Otherwise, we can decompose e as $\{s^v, X = e_1, s\}$, where e_1 is not a value. By typing, $\emptyset \vdash e_1 : \tau_1$, so by induction hypothesis, as e_1 is not a value, we have $e_1 \to e_1'$ for some e_1'. But then, $e \to \{s^v, X = e_1', s\}$ by rule CONTEXT and we are in case (b),

*(*T-STRUCT*)* If e is a structure, then it is a value and we are in case (a).

*(*T-COMPOSE*)* Then $e = (e_1 \gg e_2)$.

- If e_1 is not a value, then by induction hypothesis, we have $e_1 \to e_1'$. But then, $e \to e_1' \gg e_2$ by rule CONTEXT, so we are in case (b).
- Otherwise, if e_2 is not a value, then symmetrically, by induction hypothesis, we obtain the desired result.

- If both e_1 and e_2 are values, then by typing, there exist m_1 and m_2 such that $e_1 = \langle m_1 \rangle$ and $e_2 = \langle m_2 \rangle$, and we have a derivation of the shape

$$\frac{\dfrac{\emptyset \vdash m_1 : {M^e}_1}{\emptyset \vdash \langle m_1 \rangle : \langle {M^e}_1 \rangle} \qquad \dfrac{\emptyset \vdash m_2 : {M^e}_2}{\emptyset \vdash \langle m_2 \rangle : \langle {M^e}_2 \rangle}}{\emptyset \vdash \langle m_1 \rangle \gg \langle m_2 \rangle : \langle \mathit{Sig}(\mathit{Add}({M^e}_1, {M^e}_2, \varepsilon)) \rangle}$$

  But then, by lemma 10, $\mathit{Add}(m_1, m_2, \varepsilon)$ is defined, so e reduces by rule COMPOSE.

*(*T-CLOSE*)* Then $e = \mathsf{close}\ e_1$.

- If e_1 is not a value, then by induction hypothesis, it reduces to e_1'. But then, $e \to \mathsf{close}\ e_1'$, and we are in case (b).
- Otherwise, by typing, there exists m^c and a complete enriched signature M^e, such that $e = \langle m^c \rangle$, $\emptyset \vdash m^c : M^e$ and $\tau = \{\lceil M^e \rceil\}$. But as m^c is complete, e reduces by rule CLOSE.

*(*T-DELETE*)* Then $e = {e_1}_{|-X}$. If e_1 is not a value, then, as above, we obtain the desired result by induction hypothesis. Otherwise, e reduces by rule DELETE.

*(*T-SELECT*)* Then $e = e_1.X$. If e_1 is not a value, then, as above, we obtain the desired result by induction hypothesis. Otherwise, by typing, there exists s^v such that $e = \{s^v\}$ and $X \in \mathit{dom}(s^v)$, so e reduces by rule SELECT.

*(*T-LETREC*)* Then $e = \mathsf{letrec}\ b\ \mathsf{in}\ e_0$, and e reduces by rule LETREC.

*(*T-LET*)* Then $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. If e_1 is not a value, then we proceed as above. Otherwise, e reduces by rule LET.

End proof

Theorem 1 (Soundness) *The evaluation of a closed, well-typed expression either does not terminate, or reaches a value.*