Call-by-value mixin modules

Tom Hirschowitz

March 21, 2003
Contents

I Modularity and mixin modules 15

1 Modularity and code reuse 17

  1.1 Motivation ........................................ 17
  1.2 Safety versus flexibility .......................... 17
    1.2.1 The diamond import problem ................. 17
    1.2.2 The extensibility problem .................... 17
    1.2.3 The modifiability problem .................... 17
    1.2.4 The recursion problem ........................ 17
  1.3 An overview of mixin modules .................... 17
    1.3.1 Mixin modules ................................ 18
    1.3.2 An extended binding construct ............... 20
    1.3.3 Typing issues ................................ 20
    1.3.4 What is a mixin module not? ................ 21

2 A brief history of mixin modules 25

  2.1 Jigsaw ............................................. 25
  2.2 Duggan and Soureils’ mixin modules ............ 28
    2.2.1 Overview of the language .................... 28
    2.2.2 Expressiveness and limitations .............. 29
  2.3 Units ............................................. 32
    2.3.1 MzScheme ..................................... 32
    2.3.2 Theory ....................................... 33
    2.3.3 Types ....................................... 37
  2.4 CMS .............................................. 39
    2.4.1 Syntax and semantics ......................... 39
    2.4.2 Types ....................................... 41
2.4.3 Expressiveness and inconvenient

2.5 The m-calculus

2.5.1 Syntax and semantics

2.5.2 Expressiveness and inconvenient

2.5.3 Comparison with CMS

II Dynamic and static semantics of call-by-value mixin modules

3 Dynamic semantics: the MM language

3.1 Syntax

3.2 Semantics

4 Static semantics

4.1 Type system

4.2 A theory of dependency graphs and degrees

4.3 Graph soundness

4.3.1 Modeling the reduction with graphs

4.3.2 Subject contraction for graphs

4.3.3 Manipulation of recursive bindings

4.4 Soundness

5 Refined static semantics: type components

5.1 The MML language

5.2 Type system

5.2.1 Well-formedness

5.2.2 Typing

5.2.3 Subtyping

5.2.4 Type equivalence

5.2.5 Directions for a proof of soundness

5.2.6 Undecidability, principal types, syntactic types

5.3 Polymorphism and datatypes

5.3.1 Polymorphism

5.3.2 Datatypes

5.4 Examples
III  Compilation of mixin modules  

6  Typed compilation without local definitions  

6.1  Intuitions  
6.2  Definition of the compilation scheme  
6.2.1  Restricting the source language: MML  
6.2.2  The target language $\lambda_e$  
6.2.3  Compilation scheme  
6.3  Type soundness of the translation  
6.3.1  A type system for the target language  
6.3.2  Soundness of the target language  
6.3.3  Soundness of the translation  

7  Compilation of let rec  

7.1  Overview  
7.2  The source language $\lambda_e$  
7.2.1  Syntax  
7.2.2  Structural equivalence  
7.2.3  Semantics  
7.3  The target language $\lambda_{alloc}$  
7.3.1  Structural equivalence  
7.3.2  Semantics  
7.3.3  The $\lambda_{alloc}$ calculus and its confluence  
7.4  Translation  
7.4.1  Generalized contexts in $\lambda_{alloc}$  
7.4.2  Definition of the two translations  
7.4.3  Relating the two translations  
7.5  Correctness  
7.5.1  Translation of contexts and compositionality
7.5.2 Translation of access .............................................. 186
7.5.3 Translation of internal merging ................................. 187
7.5.4 Simulation .......................................................... 189
7.6 Related work ......................................................... 194

8 Untyped compilation with local definitions ........................... 197
A faire

Graph subtyping  In the rule \( \text{Wf-Mixin} \), it is not necessary to check that the graph has sources in \( I \) and targets in \( O \), since more edges would not break safety.

Generativity  What happens if datatype and record type declarations are not generative? Is this the end of functional languages? Answer of Xavier: Harper et al. believe so, I don’t, I rather believe that abstraction is important.

Generativity seen as effectful static operation  Can the effects of Dreyer et al. be related to Sewell’s interpretation of abstract types with \( \nu \) quantifiers? Types should be computed and \( \nu \) really has the effect of creating a new variable that can be extruded to the top level. A question is where exactly should the \( \nu \) computations be suspended, i.e. what are the lambdas for \( \nu \), under which nothing happens? Generative functors for sure, at least. Unfortunately, Sewell’s interpretation probably does not account for applicative ones anyway.

On Duggan and Soureli  The method of Duggan and Soureli for proving the soundness of their mixin modules could cause problems with recursive types, since nothing prevents type definitions from being recursive. But in fact, only datatypes can be mutually recursive, so there are only iso-recursive types.

Other problems in Soureli’s masters thesis. The syntax does not mention the inner keyword. Type strengthening is undefined on mixin module types, and wrong on functor types (corrected in the papers). Signature constraint is present in the syntax, but not in the typing rules (corrected in the paper), nor in the dynamic semantics (implicitly eluded because subj-red holds for the language without abstraction).

Type inference for MML  Polymorphism is not accounted for here. Type inference would not be satisfactory, without changing a bit the syntax. We conjecture that grouping outputs in single / block definitions in the correct order solves the problem.

Xavier’s leitmotiv about recursive modules  Lacks of expressiveness in Dreyer, Crary, and Harper’s theory:

```plaintext
module rec A : 
  sig
    type t
    val f : B.t -> t
  end
= struct
  ...
```

7
end

and B :
sig
type t
val x : t
val y : A.t
dee

= struct
type t = int
let x = 1
let y = A.f 2
dee

This program is ill-typed since during the type-checking of $y$, there is no way to identify $t$, int and B.t. Mixin modules provide a way to code such programs in a more flexible way.

**Encoding labeled and optional arguments**  Labeled argument can be encoded in any mixin module calculus in $\{CMS, m, CMS_r, MM, MML, \ldots\}$, as follows (here in MM syntax). A function expecting $n$ arguments $x_1 \ldots x_n$, labeled $l_1 \ldots l_n$, and returning the result $e$, can be represented by a mixin of the shape $(l_1 \triangleright x_1 \ldots l_n \triangleright x_n; RES \triangleright e)$. Labels are mandatory in function applications $(e\ l_1; e_i, \ldots l_n; e_i)$, which are encoded as

$$(\text{close}(e + (e_i; e_i, \ldots l_n; e_i))).RES$$

Optional arguments are added to the encoding by replacing composition with overriding in the encoding of function application, and putting the default arguments in the function, with the corresponding labels.

**Subtyping mixin modules in a mobile code scenario: MoMiMo**  The nightmare paper by Bettini, Bono, and Venneri [10] on depth subtyping for mixins in a distributed setting turns out trivial with mixin modules. The problem with mixin classes is that their types do not take the contravariance of methods into account. But it exists indeed: coercing a method specification to a super type may be unsafe, because other methods may need the more precise typing. For instance, assume a mixin class with two methods $f$ and $g$ of types $\tau$ and $\tau'$, respectively, and assume $g$ needs $f$ to be of type $\tau$ at most. Covariant subtyping of methods can lead to giving $f$ a type $\tau''$, super type of $\tau$. But then, $f$ can be overridden by a method of type $\tau''$, which makes the implementation of $g$ unsound. With mixin modules, the input type of $f$ is subtyped contravariantly, so this problem does not appear. Moreover, if subtyping points are clearly identified, as in MoMi, a mechanism of implicit coercions allows to solve the issue with mixin modules, quite straightforwardly. Where a mixin module is expected at type $\tau$, insert a coercion to type $\tau$. Graph subtyping and inputs subtyping can be assumed to be implicit, since they have no incidence on the runtime. A coercion to type $(I:O; \rightarrow)$ of any mixin module $e$ is implemented by $(e + (I; e))\_\text{dom}(O)$. It adds the missing inputs at the right types, coerces the present ones to the right types, and hides the unexpected outputs.

**Other possible designs (in future work section)**  The thesis explores the solution of definition reordering, but one could imagine a more restrictive, but perhaps more intuitive design where a mixin module is a structure with holes, where definitions cannot be reordered. Composition then attempts to just fill the holes, in a deterministic way. The idea would be: as definitions know exactly their place in the mixin module, maybe it is not necessary to include dependency information in types.
Extension of MML with additional type expressions In the style of Odersky et al. [60], it would probably be beneficial to MML to feature type expressions such as $M_1 + M_2$ and p.type and close M.

Efficiency Splitting the close operator into a reordering operator and an instantiation operator allows to perform reordering only once.

Extension of the result on letrec Ajouter une construction $block(e, n)$ au calcul $\lambda_e$ t.q.

- $block(\Box, n)$ est un lift context et un strict context,
- $\text{Size}(block(e, n)) = n$,
- $block(v, n) \to v$, pourvu que $\text{Size}(v) = n$,
- $[block(e, n)] = [e]$, mais on perd la comptitude, probablement.

References exemples Faire pointer les exemples du chapitre compil vers la section overview, des qu’elle sera prete.

Order of evaluation Abstraire sur la fonction pour trouver un ordre d’évaluation correct etant donne un graphe et les formes serait une bonne idée. On peut donner la fonction actuelle et la fonction qui ne depend que du graphe en exemple, et dire dans la compilation qu’on choisit la deuxieme pour ce chapitre.

Modularizing the proofs Modulariser la preuve de surete sur les regles d’Ariola, notamment au niveau des dependances, clarifierait grandement le rapport entre les preuves de surete de MM et de $\lambda_e$. De toute facon, pour $\lambda_e$, faut la refaire, a cause de ces regles justement.

Headers Ajouter des headers, ça fait vachement mieux.

Separate compilation A paragraph explaining how to handle separate files as closed mixins, and linking as a mixin composition followed by a closure.

Name spaces In some future work section, discuss the possibility of explicit name spaces inside mixin modules. The idea is to have semantic sub-modules, but with less rigid boundaries. In particular, closing a mixin modules containing name spaces would flatten them during computation, and reconstruct them when building the final module. Thus access in name spaces works the same as for modules, but dependencies can be finer. Motivation is found in the first attempt to implement recursive polynomials.

Notations Throughout the thesis, side-conditions are written as premises for readability. [?] means “please find a correct bibtext entry for this before giving the thesis to the referees” or “please verify this information ...”. Insérer au premier endroit ou c’est utilis la notation $\perp$ pour les ensembles disjoints, la notation $| \ldots |$ pour le cardinal d’un ensemble et la longueur d’une liste. Remplacer presque partout “variable” par “identifier”? La fiche est parse droite. On utilise plutôt la syntaxe Ocaml que la syntaxe SML. Substitution $\{x \mapsto y\}$ signifie que $x$ remplace $y$. Meta-galits : $=_{\text{der}}$ signifie “is defined as” et $\equiv$ signifie “is syntactically equal to”. Homognit des mots-clefs dans la section type-components et overview mixins. Quand est-ce que j’ai supposé qu’il y a des types produits, trouver et le dire. Faire plusieurs passes pour vérifier que ces conventions sont respecte. Remplacer les $\backslash \backslash \text{indent}$ par $\backslash \backslash \text{linebreak}$. Priorités: décrire au dbut les priorits implicites, notamment la section . a priorit sur tout.
A lire absolument  Drossopoulou, Morrisett, Machkasova, Jones, Odersky, Parnas, Szyperski.
Introduction: linguistic constructs for code reuse

The increasing size and complexity of programs cause important pragmatic industrial problems. Maintenance becomes a full time task, sometimes almost unmanageable, and safety or correctness often happens to be impossible to prove. At the same time, more and more formerly human jobs are done electronically, and it therefore becomes more important that programs really do what they are expected to. An airplane pilot program, an underground driver program, or to a least extent a train booking program, have to be correct. A natural idea to solve this problem – and it essentially was born centuries ago – is to divide problems into smaller, easier to solve ones, and to exploit and share the results. In software engineering, this can be done at several levels.

Language abstractions A first level is provided by various forms of abstraction in the considered language. As defined by Leroy [51],

“Modularization is the process of decomposing a program in small units (modules) that can be understood in isolation by the programmers, and making the relations between these units explicit to the programmers.”

Modules, functors, or classes for example, offer a way to modularize programs. But functions, or extensible datatypes, may perfectly be seen as modularization constructs. Functions, for instance allow to write code only once, whereas it otherwise ought to be inlined at every place of use. This level has been and is still being widely explored by the programming language research community. However, it only promotes code reuse at the level of one program. With only functions or objects, one cannot reuse any code from another program, whereas different programs often need the same kind of functionalities, such as graphical interface tools. Sharing such code between them requires switching to the level of separate compilation.

Libraries From [51] again,

“Separate compilation is the process of decomposing a program in small units (compilation units) that can be type-checked and compiled separately by the compiler, and making the relations between these units explicit to the compiler and linker.”

If a program is divided in several compilation units, some of these may be put in a repository, from where other programs can use them. Such compilation units are usually called libraries, and provide means of reusing code across programs. Nevertheless, this does not allow full code reuse yet, because each kind of program has its particular best programming language. If graphical interfaces are often written in object-oriented languages, this may not be the case for CPU intensive probabilistic simulations for instance. However, a simulation program would be perfectly wrapped in a graphical interface.

Components The idea of the third and last level is to allow that, and even more, to allow it across different sites. As advocated by McIlroy at the 1968 NATO conference [57], programs should be mainly built by assembling off-the-shelf components – supplied by a software components
industry, without having to modify their source codes. (This is often called “black box”.)
The component-based approach bases on two main ideas.

- First, different parts of a program may be written in different languages, keeping some
  sort of compatibility between them, thanks to an common interface definition language.
- Second, components are accessible by various ways, including the internet. The program
  may call procedures defined in a remote component, and even ask for some kind of
  components more or less automatically.

In a component approach, critical parts of programs may be written in a very fast language,
whereas the user interface, or communication parts for example, can be written in a more
expressive – or even dedicated – high-level language. More than that, the program may rely
on previously written components, without having to bother with their locations or imple-
mentations. Nevertheless, safety properties of whole programs are difficult to prove, since it
requires the ability to analyze programs in different languages within the same framework,
and to model the protocols for accessing remote components. Eventually, as a matter of
fact, most component architectures are more or less object-oriented (see e.g. [49]), in that a
component looks very much like a class. This causes problems when writing components in
languages with drastically different programming paradigms, such as functional languages.

These increasingly ambitious proposals are very promising for what concerns code reuse and reduc-
tion of program sizes, but one has to consider them with respect to safety. The first level has been
extensively studied from this standpoint, specifically through the use of sound type systems: there
are well-known ways for ensuring statically (i.e. at compilation time) that an object-oriented or
a functional program will not crash (see e.g. [50, 78, 1]). The second level has been investigated, and
sound type systems have been set up, which are able to statically prove that a separately compiled
program will not go wrong [51, 40, 19, 56, 65, 70]. The component approach is its early phase of
formalization [69], and types or safety seem to hardly be under consideration yet.

We are concerned with the first and second level, mainly. The work on designs for safe separate
compilation [51, 40, 56] has lead to introduce linguistic constructs for considering compilation units
as special datastructures, called modules. Modules are therefore a bit ubiquitous, because they may
be seen either as language constructs, almost exactly as say, functions, or as a kind of interface
between the program and the real world, here the operating system. In the OCaml language
[55] for instance, compilation units are considered exactly as modules. In SML, they are closer
to structures (the contents of a module). A consequence is that a language with modularization
constructs is a language featuring separate compilation, provided the considered constructs support
it. This allows studies of linguistic modularization constructs supporting separate compilation to
be viewed as studies about code reuse inside the considered language, and therefore to be notably
simplified. Indeed, instead of setting up a complicated framework where the file system, shell
commands, object files, are modeled, one may study linguistic modularization constructs, and
then argue that they support separate compilation, as done in [51, 65] for example. Two main
ideas for such linguistic modularization constructs have been explored, at least.

Classes and mixins Languages like Java [48] base their modularization process on classes and
objects. Objects are basically records, with a set of methods to operate on them. For example,
a window object would typically be a record of a position, a size and some sub-objects, with
methods moving it, showing it, etc. Classes are object generators, and the idea is that they
may be incrementally refined. Methods may be added and redefined as needed, thanks to
the complex mechanism of inheritance [48, 55]. In order to define a new class, the programer
can base on an existing class, without having to edit the initial code manually. Only the
modifications have to be written. Mixins are an extension of classes, where class extensions
are parameterized over the extended class, and thus may be applied to several base classes.
Important research has been done on such languages, and they are theoretically well-known.
However, this approach constrains the language very much: a module is a class, and all parts
of the program using this module have to be written in object-oriented style. This may
impede the efficiency, since object-oriented languages cannot pretend to compare with C on critical domains, such as large probabilistic simulations, or symbolic computation. Moreover, separate compilation for classes is rather limited, since for instance Java mutually recursive classes cannot be compiled separately. Consequently, object-oriented languages often rely on a system of packages in order to group related classes together.

**Modules** Another approach investigates modules systems. A module system wraps the programming language, the core language, with a typically second-class module language. The module language is in charge of all the glueing operations, and the core language handles real computation. Most module systems are largely independent from the underlying core language [54], thus not constraining in any way the employed programming paradigm. Furthermore, modern module systems provide abstract data types, thus allowing for full abstraction over implementation details, and guaranteeing that invariants of a module are not broken outside it. The main drawback of module systems is their lack of flexibility. There is a tension between the need to preserve safety and the convenience of being able to write programs according to the intuition. Early module systems such as the one of C are unsound, and lack parameterization, since they entirely rely on the file system. But even modern and sophisticated module systems, such as the one of ML, severely limit the programmer’s intuitions, for instance in not allowing mutually recursive definitions to span module boundaries.

None of these two ideas really seems to be the ultimate modularization concept, although both possess features that are necessary for such a concept. Classes allow very flexible incremental programming, since they allow to specialize a class without editing its source code, and only writing the modifications. Modules have the advantage to be independent of the underlying language, and to provide convenient incremental facilities.

This thesis examines an alternative, hybrid idea of modularization concept, called mixin modules. The original idea appeared in the early 90's with Bracha, Cook, and Lindstrom [17, 16, 18], and was further developed by Duggan and Sourelis [31], Flatt and Felleisen [36, 35], Ancona and Zucca [3, 6], and Wells and Vestergaard [76]. It consists in a module language - a modularization construct independent from the core language - with features for incremental programming, inspired by classes and mixins. Basically, a mixin module is a collection of named definitions and declarations. Declarations may be filled with definitions by composition with another mixin module. The definitions of one mixin module then fill the corresponding declarations of the other one, according to their names. Definitions are not statically bound to one another, and may be overridden.

The remainder of this thesis is organized as follows.

Chapter 1 describes known module languages and analyzes them from the viewpoints of flexibility and safety. A collection of features is presented, that have been considered necessary somewhere in the literature. Then, two widely used module systems are briefly recalled, which serves as a basis for discussing their respective lacks of expressiveness in the last section.

Chapter 2 summarizes previous work on mixins modules, from Bracha’s seminal thesis [16], to the latest theoretical formalizations [6, 76, 45].

Chapter 3 defines MM, our language of mixin modules, and its operational semantics. The semantics is defined thanks to the introduction in the language of a new construct let rec for binding mutually recursive definitions, which is more general than most such other ones.

Chapter 4 presents and proves sound a simple type system for MM, dealing with the recursion problem in an elegant manner.

Chapter refsection-implementation elaborates an implementation strategy for the let rec construct. Its presentation abstracts over the implementation of the rest of the language.

Eventually, chapter 8 examines the remaining problems and ideas for solving them.
Part I

Modularity and mixin modules
Chapter 1

Modularity and code reuse

1.1 Motivation

Encapsulation, abstraction, hierarchy, ... cf Bracha, Wells, Harper, ...

Transition: a consequence of modularization is that programs (or programmers if the constructions
for modularization are extra-linguistic) have to perform the assembly operations to construct the
intended code out of pieces. These operations are subject to failure, and it is difficult to set up
sound type systems for all of them. As a consequence, more flexible modularization constructs
(such as objects or components) provide less abstraction mechanisms and safety properties that
more sound ones (such as ML modules).

1.2 Safety versus flexibility

1.2.1 The diamond import problem

1.2.2 The extensibility problem

cf Flatt...

1.2.3 The modifiability problem

cf FOC

1.2.4 The recursion problem

cf ML

1.3 An overview of mixin modules

A characterization of mixin modules In [6], Ancona and Zucca give a semantic characteriza-
tion of a system of mixin modules, in terms of a characterization of module systems, and some
requested features.
Definition 1 (Module system) A module system is a language dedicated to modularization, built on top of a core language, and meeting the two following requirements.

- First, the module system must be as independent as possible from the core language. Ideally, it can be instantiated over several core languages, in a systematic way.
- Second, a module should correspond to a compilation unit, thus providing for separate compilation.

Typically, module languages are expected to feature parameterization (the ability to use a module in different contexts). Then, a mixin module system is defined as module system providing two particularly important features for modularization.

Definition 2 (Mixin modules) A module system supports mixin modules if it supports cross-module recursion and overriding.

Presentation by example [Maybe split this in: here, example hiding the problems with let rec, moving them to an overview subsection in the section on MM]

1.3.1 Mixin modules

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

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

```
mixin A =
  import
  val x : int
  val f : int -> int
export
  define y = (g 0) + x
  define g z = ... f ...
end
```

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

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

```
mixin B =
  import
  val y : int
  val g : int -> int
export
  define x = y + 1
  define f z = ... g ...
end
```
The composition mixin $C = A + B$ of $A$ and $B$ is equivalent to the mixin module:

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

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

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

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

mixin D = import
  export
    define x = 0
  end

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

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

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

module $\mathbb{M} = \text{struct}
  let rec g z = ... f ...
```

19
\[f \ z = \ldots g \ldots\]

let \(x = 0\)

let \(y = (g \ 0) + x\)
end

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

module \(M =\) struct
  let rec \(f \ z = \ldots g \ldots\)
  and \(g \ z = \ldots f \ldots\)
  let \(x = 0\)
  let \(y = V\)
end

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

We refer to [16, 6] for more details on mixin modules and other operators.

### 1.3.2 An extended binding construct

In \(MM\), the definitions of \(x\) and \(y\) could not have been included in the mutually recursive definition of \(f\) and \(g\). Indeed, the \texttt{let rec} construct of ML only allows to bind syntactic functions (or constructed values in the case of OCaml). Therefore, in the case of more complex dependencies between the definitions of a mixin module, instantiation would lead to nested \texttt{let} and \texttt{let rec} bindings. In order to avoid this complication, our calculus features a slightly more powerful \texttt{let rec} than that of ML, which is reminiscent of monadic recursive bindings [33]. It evaluates the definitions from left to right, and basically only goes wrong when the value of a variable defined to the right of the current definition is needed. For instance, the definition

\[
\text{let rec } f \ x = \ldots g \ldots \\
\quad g \ x = \ldots f \ldots \\
\quad x = 0 \\
\quad y = (g \ 0) + x
\]

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

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

### 1.3.3 Typing issues

Our \texttt{let rec} is not much more powerful than that of ML. Its main interest is that complex series of sequential \texttt{let} bindings and mutually recursive \texttt{let rec} bindings are now written as straightforward definitions. Its typing is much less straightforward of course, since it requires the analysis of dependencies between the definitions. This analysis has to go beyond immediate dependencies, as shown by the following example.
Example 1 Consider the following binding, where braces enclose records and \texttt{x} and \texttt{z} are record field names.

\begin{verbatim}
let rec x = { X = z } 
y = x.x.Z 
z = { Z = 0 }
\end{verbatim}

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

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

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

1.3.4 What is a mixin module not?

A functor

There are facilities to extend an existing functor with new fields. However, this kind of extension differs in at least two important ways from the way a mixin extends another mixin.

First, existing fields will be shadowed by new definitions with the same name. With mixin modules, depending on the operator used for the extension, a previous field with the same name either yields a clash or is overridden. In other words, mixin components are late-bound together, whereas module components are statically bound.

As an example, consider the following module:

\begin{verbatim}
module A =
  struct
  let f x = x
  let x = f 0
  end
\end{verbatim}

If we try to extend it with a mixin module, we define:
mixin B =
import
export
  define f x = x + 1
end

And then, we compose the two mixin modules by overriding (A <- B) to obtain a new mixin module equivalent to

import
  define x = f 0
  define f x = x + 1
end

The previous value of f has been removed, and instantiating the result yields a module equivalent to

struct
  let f x = x + 1
  let x = 1
end

If conversely we try to extend it with a functor, we rather write:

module B' (X : sig val x : int end) = struct
  include X
  let f x = x + 1
end

Then, we apply the functor (B'(close A)). This time the result is equivalent to

struct
  let f x = x
  let x = f 0
  let f x = x + 1
end

which evaluates to

struct
  let x = 0
  let f x = x + 1
end

Another difference between functors and mixin modules is that, a mixin module really extends something, whereas a functor could rather be said to coerce it first, and then extend the result. Indeed the argument of a functor is ascribed a signature, and during functor application, is coerced to this signature. As a result, if an argument with more fields than expected is passed as an argument to a functor (that extends its argument), the result will not mention the unexpected fields.

Consider for example the functor
module F(X : sig end) = struct
  include X
end

and the mixin module

mixin A =
  import
  export
end

When applied to a module \( X = \text{struct let } x = 0 \text{ end} \), the functor \( F \) generates an empty module, thus not extending \( X \) at all.
On the contrary, when composed with a mixin module \( X = \text{import export define } x = 0 \text{ end} \), the mixin module \( A \) evaluates to a mixin module equivalent to \( X \), thus really re-exporting all the components of it.

A mixin class

A mixin class is basically a class extension parameterized over the superclass it extends. It is a special kind of function over classes. At first glance, a first difference appears: mixin classes are tied to the object-oriented programming paradigm. True, but not enough to make a clear distinction: a class exports some definitions, as a module does, and field definitions require some computation to happen at initialization time, which is pretty much the kind of interaction mixin modules have with their clients.

A deeper difference is that (at least our) mixin modules feature component reordering according to their dependencies, thus allowing to automatically rearrange almost any kind of program parts. On the class side, no reordering of initialization computations is performed, so mixin classes are less expressive in this respect. Further, mixin modules allow to specify the order in which computations are performed, which is not the case with classes.

According to Bracha [16], mixin module operators express inheritance mechanisms in a finer way than mixin classes operators. Specifically, they allow to resolve conflicts during multiple inheritance more flexibly than with mixin classes. However, since then, new notions of mixin classes have appeared, which could invalidate this claim [35].

On the whole, the underlying fundamental idea of any module system is conservativity. Conservativity is a semi-formal term designating the property that any running code obtained by combination of modules could have been produced without the module system, by a monolithic program. This notion concerns the structure of the program as well as the efficiency, and it requires that, as far as possible, a module system only does modularization, and that it does not decrease the overall efficiency of programs. Mixin class-based systems are extensions of class-based systems. In practice, it implies object-oriented programming, so mixin classes as a module system do more than just modularize some monolithic code. In theory, one could use mixin classes as a pure modularization construct, but as such they are not expressive enough. Indeed, they hardly allow more than grouped, parameterized definition of functions with late binding and inheritance. This is quite powerful already, and roughly corresponds to Jigsaw plus initialization (see section 2.1), but is not flexible enough with respect to initialization. In programs, computations can be arbitrarily interleaved with function definitions, and this should be reflected by the module system.

On the contrary, some class-based languages, such as C++ [73], allow to share some data between all instances of a same class, through “static members” definitions, which mixin modules do not support directly.
Chapter 2

A brief history of mixin modules

2.1 Jigsaw

In his PhD thesis [16] and the related articles with Cook [17] and Lindstrom [18], Bracha presents for the first time the idea of mixins as a modularity mechanism relatively independent from the base language – the Jigsaw "framework". His mixins are partially defined records of named values under a recursive scope. They are equipped with a set of novel operators on them, which express in a very clean way multiple inheritance with enhanced flexibility, component sharing, renaming, hiding, and redefinition (overriding).

We briefly give an idea of the language. Its syntax is in figure 2.1. We abstract over typing issues. A module is a sequence of comma separated declarations (a label $X$), and definitions $def$, which can be core definitions $X = E$ or module definitions $X = e$. Definitions must bind values, and may refer to one another and to declarations. Modules must be closed: they must not have any free variable. This is really simple yet: no virtual definitions, no distinction between instance variables and methods, no "friend" declarations, etc. The complexity and expressiveness of Jigsaw resides in the operator suite.

Module composition $|\!|\!|$ has the effect of filling the holes (the declarations) of both modules with the definitions of the other, and vice versa. For example, if label $X$ was declared in $e_1$ and defined in $e_2$, it is now defined in $e_1 |\!|\!| e_2$, as in $e_2$. Modules must not have any definition in common, only declarations, possibly.

Module overriding is similar, but modules may define common labels. The ones from the right argument replace the ones from the left argument. Definitions are late bound by default: assume that in $e_1$, the definition of $X$ makes a call to $Y$, and that $Y$ is defined in both $e_1$ and $e_2$. Then, in any instance of $e_1 \leftarrow e_2$, the one and only $Y$ definition available is the one from $e_2$, and $X$ calls it as well.

There is a way to make binding static, through the freeze operator. After freezing a label $X$ in a module, it is still available to the outside world, but the other definitions of the module semantically rather refer to a local copy of it, which cannot be modified anymore. The dual operator freeze_all_except freezes all labels but the given ones.

Name conflicts during a composition $e_1 |\!|\!| e_2$ may be solved in several basic ways. Assume for example that the label $X$ is defined in both $e_1$ and $e_2$.

- If one of the conflicting definitions, say the one of $e_1$, must be chosen as the final one for both modules (overriding the one of $e_2$), then $X$ may be deleted from $e_2$ ($e_2 \setminus X$). This can be done another way, in the case where only a few definitions have to be kept from one module,
Module: \[\begin{align*}
e & ::= \text{module } binding_1, \ldots, binding_n \text{ end} \\
& | e_1 \parallel e_2 \\
& | e_1 \leftarrow e_2 \\
& | e \text{ freeze } X \\
& | e \text{ freeze all except } \{X_1 \ldots X_n\} \\
& | e \mid X \\
& | e \pi X_1 \ldots X_n \\
& | e \text{ hide } X \\
& | e \text{ show } \{X_1 \ldots X_n\} \\
& | e[X_1 \mapsto X_2] \\
& | e \text{ copy } X \text{ as } Y
\end{align*}\]

Basic module
Composition
Override
Freezing
Complementary freezing
Deletion
Projection
Hiding
Show
Renaming
Copy as

Instance: \[i ::= \text{instantiate } e\]

Definition sequence: \[\begin{align*}
\text{binding} & ::= X \mid \text{def} \\
\text{def} & ::= X = e \mid X = E
\end{align*}\]

Binding
Definition

Figure 2.1: Syntax of Jigsaw

by the projection operator, which deletes all definitions of a module, except the given ones \((e_i \pi X_1 \ldots X_n)\).

- If one of them, say \(X\) in \(e_1\), is the good one for the outside world, but the definition of \(X\) in \(e_2\) must still be referenced by the definitions of \(e_2\) after composition, then \(X\) can be hidden in \(e_2\) (\(e_2 \text{ hide } X\)). Other components will keep their anonymous copy of it, but it will not appear in the interface of the module, which may safely be merged with \(e_1\). Similarly to deletion, this can be done another way, in the case where only a few labels have to remain visible, by the show operator, which hides all components but the given ones.

- If conflicting names have to be kept both, having different capabilities, then definitions and declarations may be renamed \((e_i[X \mapsto Y])\). The new names must not be mentioned in the argument.

Of course these operators may be used for different purposes; for example, renaming may be used for plugging a definition of a module in an input label of another module, even if they do not initially have the same names.

The last operator, copy, is not easy to understand; especially, it is not easy to see why it is important. Bracha takes the example of a mixin supposed to add borders to windows, in the context of a window manager, and of an object-oriented core language. Let \texttt{Border} be a mixin defining the functions \texttt{display} and \texttt{display\_border}, and declaring only the missing function \texttt{display\_body}. The function \texttt{display} successively calls \texttt{display\_border} and \texttt{display\_body}.

\begin{verbatim}
Border = module
    display\_body, display () =
        display\_border C ;
        display\_body C
end
\end{verbatim}

Now we dispose of another mixin defining the functions for windows. Assume it has been defined as follows.
Window = module
    display () =
    ...
end

The reasonable intention is to plug window.display in border.display_body, keeping the name border.display. So a kind of renaming of display into display_body in window is needed. But it is more complicated than that. Other references to display inside window should refer to the final display function, i.e. border.display, and that would certainly not be the case with renaming; they would point to the renamed window.display function. The expression copy display as display_body copies the body of display, giving it the label display_body. It is then possible to override display with the definition from border, obtaining the expected behaviour.

Conclusion From the standpoint of design, Jigsaw is rather convincing: mixins are more powerful than classes, without their traditional problems with binary methods, or multiple inheritance for example. They move the expressive power from the basic construct to the operator set, resulting in a more flexible design. But more generally, mixins are presented as being usable “to introduce modularity into a variety of languages, regardless of whether they support first class objects”. More than that, it suggests that highly expressive mechanisms for modularization such as inheritance, late binding, and so on, could be exploited outside the context of object-oriented languages. Indeed, one of the main characteristics of objects is self application, which is not at all a necessity in Jigsaw. Bracha did not push this aspect of his work as much as he could have: the only examples and applications given, including a full fledged implementation of Modula-π (an extension of Modula-3 [20]), are object-oriented. The extension of non object-oriented languages is only informally suggested.

The framework has official weaknesses of course, such as the lack of support for name based typing (type abbreviations, generative types), or for abstract data types.

Other drawbacks of Jigsaw are:

- Modules do not contain any free variable.
- Modules contain only values, which almost reduces mixin-based module systems to libraries of functions. For example, if instantiation of a mixin requires some initialization code to be run, it has to be done manually, which breaks the abstraction power of mixins modules.
- The semantics of Jigsaw is given by translation to an untyped λ-calculus with records, and typing rules are given. The type system is pretended to be sound, but there is no attempt to argue on that, and doubt remains about some oddities. For example, in the informal description, mixins look like classes, with a global recursive scope, and in the formal definition, they are translated as records of values. This leads to assume that the source code of modules will be translated as self access to components, by record label, but this does not appear at all in the translation.

From the standpoint of implementation, interesting ideas are introduced, especially the notion of dynamic and static offsets for compiling method calls. The implementation of the freeze operator though, seems strange, since it does nothing, as if freezing would only act on the type system. Nevertheless, it seems that this does not fit the semantics.

As a conclusion, Jigsaw is not really usable directly as a sound theoretical basis for mixin modules, but its design must be a guideline for our investigations.
<table>
<thead>
<tr>
<th>Core identifiers</th>
<th>Module identifiers</th>
<th>Type identifiers</th>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in \text{Vars}$</td>
<td>$s \in \text{MVars}$</td>
<td>$t \in \text{TVars}$</td>
<td></td>
</tr>
<tr>
<td>$x \in \text{Names}$</td>
<td>$s \in \text{MNames}$</td>
<td>$t \in \text{TNames}$</td>
<td>$c \in \text{CNames}$</td>
</tr>
</tbody>
</table>

Path: \[ p ::= \epsilon | s | p.s \]

Module: \[ M_t ::= \text{struct } M \text{ end } | \text{functor}(\text{structure } s_k : S_t) M_t \ | (M_t : S_t) \\
| p | p(p) | M \zeta(M_b) M | M_t \otimes M_t | \text{clos}(M_t) \]

Mixin body: \[ M_b ::= \text{val } f \lambda x. E \ | \text{type } t\_k | t\_k | s \in M_t | M_b | \epsilon \]

Module body: \[ M ::= \text{structure } s_k = M_t \ | \text{type } t | t = \tau | t\_k \in \Phi | M ; M ; M | D \]

Value definition: \[ D ::= \text{fun } f^1 = \lambda x_1. E_1 | \ldots | f^2 = \lambda x_2. E_2 \\
| \text{val } x_k = E \ | D ; D | \epsilon \]

Core: \[ E ::= \bot | x \ | p. x \ | p. c(E_1 \ldots E_n) \ | E \ | E \ | \lambda x.E \\
| \text{let } D \text{ in } E \ | \text{case } E \text{ of } \text{R}_1 | \ldots | \text{R}_n \ | \text{inner} \]

Matching: \[ R ::= P \Rightarrow E \]

Pattern: \[ P ::= x \ | p. c(x_1 \ldots x_n) \]

Module type: \[ S_t ::= \text{sig } S \text{ end } | \text{fun} \text{sig}(\text{structure } s_k : S_t) S_t \ | S \zeta(S) S \]

Signature: \[ S ::= \text{type } t | t = \tau | t\_k \in \Phi | \text{val } x_k \ | \tau \\
| \text{structure } s_k : S_t \ | S_t ; S | \epsilon \]

Core type: \[ \tau ::= t | p. t | \tau_1 \rightarrow \tau_2 \]

Data type definition: \[ \Phi ::= c(\tau_1 \ldots \tau_n) | \Phi \cup \Phi \]

Figure 2.2: Syntax for DS

2.2 Duggan and Sourelius’ mixin modules

In [31, 32], Duggan and Sourelius introduce a language of mixin modules, which we will name DS here. It is a proposal for making ML modules more extensible. Their work is quite different from Bracha’s: they do not attempt to use his operators, except composition and instantiation. A consequence is that their mixin modules do not feature renaming, deletion, or copy. However, they feature a more powerful – and more specific to ML – version of composition. Indeed, when two mixin modules $A$ and $B$ define and export a function $f$, and when this function is defined with pattern-matching in both mixin modules, then the composition of $A$ and $B$ attempts to merge those pattern-matchings, thus building a less partial $f$ from the two initial ones. There are similar mechanisms at the level of concrete data types, building a new data type with the constructors of both arguments. Also DS features a limited form of late binding, as explained below.

2.2.1 Overview of the language

More formally, DS is defined by the syntax in figure 2.2. A distinction is made between names $x, s, t$ and variables $x, s, t$, respectively for the core language, for modules, and for types. A basic mixin module $A \equiv \text{def } M_1 \zeta(M_b) M_2$ is decomposed into three parts: the prelude $M_1$, the body $M_b$, and the initialization $M_2$. The module bodies $M_1$ and $M_b$ are usual ML structure bodies: they are lists of named definitions, including any core language expression, mutually recursive functions, modules, or types. Each definition is bound both by a name and a variable. In a mixin module, binding variables are alpha-convertible, and binding names are not, because linking mixin modules is based
on names, as we will see. In $A$, the body $M_b$ is a restricted form of structure body: only syntactic functions and ML-like data type definitions are allowed. Mixin modules can be merged together with the composition operator $\circ$. The prelude and initialization parts of the two arguments are sequenced one after another, and the bodies are merged, in the following sense.

- Two functions definitions $\lambda x. E_1$ and $\lambda y. E_2$ bound to the same external name (which are then required to have the same internal variable), are merged into one function $\lambda x. E_1 \{ \text{inner } \mapsto \lambda y. E_2 \}$

- Two data type definitions $\Phi_1$ and $\Phi_2$ bound to the same external name (which are then required to have the same internal variable), are merged into a single data type definition $\Phi_1 \cup \Phi_2$, provided no constructor names are defined twice.

Thus, the composition of two basic mixin modules $M_1^1 \zeta(M_1^1)M_1^1$ and $M_2^1 \zeta(M_2^1)M_2^1$ is

$$(M_2^1 \zeta(M_2^1) \circ M_1^1)(M_2^1 \circ M_1^1),$$

where $\circ$ denotes the above described merging. Apart from the mixin bodies, the two arguments’ components should not interact, and their binding variables are required to be pairwise disjoint.

Mixin modules contain unevaluated code, and the close operator $\text{ clos }$ allows to evaluate them, and create a proper module with their exported components. Other syntactic entries include the ones of the ML-module language, namely basic modules $\text{ struct }$ $M$ $\text{ end }$, functors $\text{ functor}(\text{ structure } s_q : S_i) M_i$ (where $S_i$ is a mixin module type). The core language is a toy functional language with pattern matching, and two special constructs:

- the $\perp$ “undefined” construct, which arises in case of an expression matched by none of the proposed patterns;
- and the inner construct, which calls a future extension of the considered function.

### 2.2.2 Expressiveness and limitations

**Main expressiveness example**

The $DS$ language allows to more intuitively modularize interpreters, and by extension any program operating on structures similar to abstract syntax trees. The idea is demonstrated by writing an interpreter for a toy language in $DS$, as sketched hereafter. The particularity of the interpreter is that it is implemented as a set of mixin modules that only have to be composed together to build the complete program, even though these mixin modules split cyclic definitions. For instance, numerical constants are treated by the following $Num$ mixin module.

\[
\begin{align*}
\text{Num} = & \text{def } \zeta \left( \begin{array}{l}
\text{type } t_{\text{term}}; \\
\text{type } v_{\text{value}}; \\
\text{type } e_{\text{env}}; \\
\text{eval}_{\text{eval}} = \\
\end{array} \right. \\
& \begin{array}{l}
t \text{is } \text{Const}(\text{int}) \\
v \text{is } \text{Num}(\text{int}) \\
e = \text{string } \mapsto v \\
\lambda x.\lambda e.\text{case } x \text{ of } \\
\quad \text{ Const}(i) \Rightarrow \text{Num}(i) \\
\quad x \Rightarrow (\text{inner } x \text{ env})
\end{array}
\end{align*}
\]

The $Num$ mixin module defines the $\text{Const}$ constructor for the type $t_{\text{term}}$ of terms, and the part of the $\text{eval}$ evaluation function which evaluates it to the corresponding $\text{Num}$ constructor, bound.
to the value type for values. The mixin module may be composed with the mixin module for functions and applications, defined by Func:

\[
\text{Func} = \begin{cases} 
\text{fun bind} & \text{bind} = \lambda x. \lambda v. \lambda env. \lambda y. \text{if } x = y \text{ then } v \text{ else } (env y) \\
\text{Val} & \underline{\text{type t_term;}} \quad t \text{ is Var} \\
& \quad \text{(string)} \\
\text{Abs} & \quad \text{(string, } t) \\
\text{App} & \quad \text{(t, t)} \\
\text{type v_value; } & v \text{ is Clos(t, e)} \\
\text{type e_env; } & e = \text{string } \rightarrow v \\
\text{eval} & \text{eval} = \lambda x. \lambda env. \text{case } x \text{ of} \\
\text{Var(s)} & \Rightarrow (env s) \\
\text{Abs(s, term)} & \Rightarrow \text{Clos(Abs(s, term), env)} \\
\text{App(f, term)} & \Rightarrow \begin{cases} 
\text{case eval f env of} \\
\text{Clos(Abs(s, f body), env')} & \Rightarrow \\
\text{eval body (bind s (eval term env) env')} \\
\text{x } \Rightarrow \text{raise Error} \\
\text{x } \Rightarrow (\text{inner x env})\end{cases}
\end{cases}
\]

Similarly, Fun define the constructors and the evaluation related to the handling of higher-order functions in the interpreted language. The composition of Num and Fun yields a mixin module equivalent to

\[
\text{Interp} = \begin{cases} 
\text{fun bind} & \text{bind} = \lambda x. \lambda v. \lambda env. \lambda y. \text{if } x = y \text{ then } v \text{ else } (env y) \\
\text{Val} & \underline{\text{type t_term;}} \quad t \text{ is Var} \\
& \quad \text{(string)} \\
\text{Abs} & \quad \text{(string, } t) \\
\text{App} & \quad \text{(t, t)} \\
\text{Const} & \quad \text{(int)} \\
\text{Num} & \quad \text{(int)} \\
\text{type v_value; } & v \text{ is Clos(t, e)} \\
\text{type e_env; } & e = \text{string } \rightarrow v \\
\text{eval} & \text{eval} = \lambda x. \lambda env. \text{case } x \text{ of} \\
\text{Const(i)} & \Rightarrow \text{Num(i)} \\
\text{Var(s)} & \Rightarrow (env s) \\
\text{Abs(s, term)} & \Rightarrow \text{Clos(Abs(s, term), env)} \\
\text{App(f, term)} & \Rightarrow \begin{cases} 
\text{case eval f env of} \\
\text{Clos(Abs(s, f body), env')} & \Rightarrow \\
\text{eval body (bind s (eval term env) env')} \\
\text{x } \Rightarrow \text{raise Error} \\
\text{x } \Rightarrow (\text{inner x env})\end{cases}
\end{cases}
\]

Here Interp is observationally equivalent to the composition of Num and Fun, but in DS, the merging of the two eval functions would rather appear as a first matching on the Const constructor, and another, nested one on the remaining constructors, replacing the initial call to inner.

Notice also that in DS, strictly speaking, the env type could not be shared during composition as in Interp, since only data type definitions are allowed in mixin bodies. A workaround would be to inline the definition of env in the mixin bodies, and possibly to export it in the initialization section. Alternatively, an extension of DS, allowing any type definition in mixin bodies, and merging type abbreviations when equal, would probably not be too difficult to formalize.

Other observations on expressiveness

Generalized abstraction As Bracha's mixins, mixin modules in DS allow to abstract over some module components in another way than with functors. Indeed, putting a definition \( f_k = \)
fun \( x \) (inner \( x \)) in the body of a mixin module \( s_1 \) has the same effect as abstracting over \( \hat{f} \). The advantage is that the mixin module \( s_2 \) providing the definition for \( \hat{f} \) could perfectly have abstracted over another definition \( g \), which \( s_1 \) would provide. However, in DS, as in Jigsaw, this abstraction mechanism does not work with mixin modules, since they are not allowed in mixin bodies. The only way to abstract over them is by functor abstraction. This makes the above example of abstraction impossible to implement directly with mixin modules instead of functions. In other terms, mixin module specific features in DS do not concern nested mixin modules.

Extension In [32], mixin modules are slightly extended with extensible data types constructors. This means that during composition, two type constructors with the same names, respectively expecting two lists of types \( \tau_1 \cdots \tau_{n_1} \) and \( \tau_1' \cdots \tau_{n_2}' \), are merged. The result is a type constructor expecting the list of types \( \tau_1 \cdots \tau_{n_1} ; \tau_1' \cdots \tau_{n_2}' \). The extended calculus is used to show how to implement interpreters for domain-specific languages in a modular way [30].

Overriding The DS language features a limited form of overriding, for components defined in the body of the considered mixin module. Indeed, a function \( f \), exported by a mixin module \( A \) can be overridden with the new definition \( E \), not mentioning inner, by composing \( A \) with the mixin module \( B = \text{def} \ \zeta(f = E) \), obtaining \( B \odot A \). The definition of \( f \) in \( B \odot A \) is \( E\{\text{inner} \mapsto E_0\} \), where \( E_0 \) is its definition in \( A \). And it is equal to \( E \), since it does not mention inner. At close time, other definitions will refer to the new definition.

Typing The DS language is equipped with a type system based on manifest types [51, 40], and featuring type abstraction. Soundness is known to be difficult to prove in the presence of type abstraction. Indeed, an expression supposed to be of an abstract type \( t \) only evaluates to a value of its implementation type, say \( \text{int} \) for example. The equational theory of types does not contain the equality \( t = \text{int} \), and therefore subject reduction does not hold. For DS, soundness is proved in a non-standard way. First, a new type system is defined, as the initial one, but without type abstraction. Basically, the types of modules in the second system are types of the first one, but are required to only export manifest types. It is then showed that a term of type \( S_i \) in the system with type abstraction is necessarily well-typed in the one without type abstraction. Finally, soundness is proved for the type system without type abstraction, which entails soundness for the one type abstraction (see also [56]). Notice that this is a proof of type soundness, in the sense that well-typed programs do not go wrong, but it does not guarantee that abstraction is preserved during reduction. Indeed, it does not prove that during reduction, values of abstract types will not be used at other types.

Conclusion

DS contains many interesting ideas for the design of a highly modular, ML-like language. However, all its features are expressed through the single composition operator. Bracha aimed at splitting the complexity of modularity into specific, simpler operators. The language DS does not follow this recommendation. Moreover, it ties the module language to the particular core language ML, and specifically to extensible pattern matching and data types. Extensible pattern matching and data types are certainly useful, but not in every case, and we prefer to consider their treatment as orthogonal to the module system. Finally, the fact that mixin module specific features are restricted to a dedicated area, where only datatypes and functions are admitted, seems a bit \textit{ad hoc}, and we would prefer a cleaner treatment.
2.3 Units

The “Programming language team” (PLT), specifically Felleisen, Flatt, and Krishnamurthi, have widely studied the subject of language designs for increasing the reusability of software components. The component-based and object-oriented approaches have been investigated, but what interests us here concerns the modular approach.

2.3.1 MzScheme

An important result of their work is of practical nature, and consists in the extension of the programming language MzScheme with units. MzScheme [34] is an implementation of the programming language Scheme, a dynamically typed, functional and imperative language, originating in Lisp. Units are a language construct dedicated to modularization. The idea comes from the observation that if packages were not hard-wired to their imports, then they would be extensible. What is intended by “hard-wired” here is that packages syntactically refer to fixed external imports. Flatt’s idea consists in making these imports abstract, i.e. parameters of the package, and making all further links between packages explicit to the programmer: if a package A provides the value f, and the package B imports a value g, and if the programmer estimates that A’s f corresponds to what B’s g is expected to do, then they may be linked together by an expression such as (simplified)

\[
\text{(compound-unit} \\
\text{(import ...)} \\
\text{(export ...)} \\
\text{(link (A)} \\
\text{\quad (B (A f)))))
\]

specifying that B’s import if filled by A’s f.

The language is designed according to Flatt’s principle of external connections [35]:

A language should separate component definitions from component connections.

In MzScheme, a unit is a completely standard data structure, resembling a record of possibly mutually recursive named definitions, and initialization expressions. The thing is that definitions can be empty; in other terms, the record has holes. Some of the record definitions may be just declared, instead of defined.

As an example basic unit, consider the following unit DB, defining a database structure, parameterized over the way the client wants to report errors.

\[
\text{(unit} \\
\text{\quad (import error)} \\
\text{\quad (export new insert delete)} \\
\text{\quad (define new \ldots)} \\
\text{\quad (define insert \ldots)} \\
\text{\quad (define delete \ldots))}
\]

Nothing special here, it resembles a function. But now, let GUI be:
\[
v ::= \text{unit} \mid c \mid \text{fn } x \Rightarrow e \\
e ::= \text{compound-expr} \mid \text{invoke-expr} \mid \text{letrec-expr} \mid e; e \mid x \mid ee \mid v \\
\text{unit-expr} ::= \text{unit import variable-mapping}^* \\
\text{export variable-mapping}^* \\
\text{definitions } e \\
\text{compound-expr} ::= \text{compound import } y^* \\
\text{export } y^* \\
\text{link } e \text{ link and } e \text{ link} \\
\text{invoke-expr ::= invoke } e \text{ with value-invoke-link}^* \\
\text{letrec-expr ::= let rec } \text{definitions } \text{ in } e \\
\text{definitions ::= value-defn}^* \\
\text{value-defn ::= val } x = v \\
\text{link ::= with } y^* \text{ provides } y^* \\
\text{variable-mapping ::= } y = x \\
\text{value-invoke-link ::= } y = e \\
\text{x ::= variable} \\
\text{y ::= linking variable} \\
\text{c ::= primitive constant}
\]

Figure 2.3: Syntax for Unit₃

(\text{unit} \\
\text{(import insert)} \\
\text{(export open error)} \\
\text{(define open ...)} \\
\text{(define error ...}))

defining the user interface for the previous database. Recursion is allowed to span unit boundaries, so DB and GUI may be connected to form a compound unit \text{PROGRAM}. As we have both DB depending on GUI through \text{error} and \text{vice versa} through \text{insert}. This solves the recursion problem from the standpoint of expressive power, but not with respect to safety, since nothing ensures that the recursion is well-founded.

Units are pieces of unevaluated code, and triggering the evaluation of a complete unit is done by the \text{invoke form}, as in \text{invoke PROGRAM}. This triggers a left to right evaluation of all the clauses in the unit body.

Units are first-class values, and this makes the language particularly expressive. In particular, units directly account for dynamic linking, since a choice between several units may be made at runtime. As a demonstration of expressive power, Flatt [35] elegantly solves an instance of the extensibility problem with units and classes, through a straightforward encoding of mixins, as units importing a class and exporting the modified class. Units do not feature overriding of definitions, and therefore a solution to the modification problem with units probably would use class inheritance for this.

\subsection*{2.3.2 Theory}

In his thesis [35], Flatt formalizes a theory of units, in three calculi. Unit₃, the first unit calculus more or less models the behavior of MzScheme. The next two ones (Unit₄ and Unit₅) successively add constructed types and type abbreviations to Unit₃.
The Unit₄ calculus The syntax for Unit₄ is defined in figure 2.3. A unit is a quadruple of a list of imports, of the shape \( y₁ = x₁ \ldots y_n = x_n \), a list of exports, of the same shape, a list of value definitions \( x₁ = v₁ \ldots x_n = v_n \), and an initialization expression \( e \). Roughly, a unit is an incomplete, unevaluated program, and it may be combined with other units, almost arbitrarily. The import and export sections serve to mediate the internal name space of the unit with its environment, through the use of linking variables. A syntactic distinction is made between linking variables, denoted by \( y \), and plain variables \( x \). Linking variables act as external names for definitions. Indeed, we will see that during the composition of two units \( A \) and \( B \), plain variables from \( A \) must be different from the ones from \( B \) (and vice versa), except if they are imported or exported as the same linking variable. A pictorial view of a basic unit is given in figure 2.4. Dotted arrows indicate a possible dependency of the target on a plain variable defined by the source: apart from free variables, the definitions are allowed to refer to themselves and to imported variables. The initialization expression is allowed to refer to both definitions, imported variables, and also external, free variables. The plain arrow requires the internal variables of the target to be included in the ones of the source: the exported variables must be defined within the unit (with arbitrary external names).

The compound construct composed two units \( A \) and \( B \) as follows.

\[
\text{compound}
\begin{align*}
\text{import } \overline{y} \\
\text{export } \overline{x} \\
\text{link } A \text{ with } \overline{y}_{\text{in}} \text{ provides } \overline{y}_{\text{in}} \\
\text{and } B \text{ with } \overline{y}_{\text{in}} \text{ provides } \overline{y}_{\text{in}}
\end{align*}
\]

The notation \( \overline{y} \) represents a sequence of linking variables. Two intermediate layers of variable mappings are introduced: the import layer \( \overline{y}_{\text{in}} \overline{y}_{\text{in}} \) and the export layer \( \overline{y}_{\text{in}} \overline{y}_{\text{in}} \). Their role is
to make connections between both arguments and with the external interface of the result. The arguments are expected to evaluate to basic unit expressions, and the semantics of composition then simply merges the sets of definitions and sequences the two initialization expressions. The difficulty is that all layers must agree on internal variables, as indicated in figure 2.5. The arrows represent inclusion of variable mappings: the target variable mapping must be included in the union of the source variable mappings. Moreover, intermediate variable mappings allow to statically have an estimation of imports and exports of both arguments, even when they are not syntactic basic units. Besides, they enable to resolve some name conflicts. Indeed, if the two arguments export a variable \( y \), it is possible to ignore one of them, by simply not putting it in the corresponding export layer. We will see below that this enables a form of subtyping.

The \textit{invoke} form transforms a unit into a let rec as expected, and the rest of the calculus, featuring functions and let rec bindings, is exactly as expected.

\textbf{Restrictions}  The calculus indeed models MzScheme, with some restrictions.

1. Packages are not modeled (thus restricting separate compilation to single units).
2. The composition operator is binary instead of n-ary and that it does not allow renaming during composition.
3. Definitions must be values.

Simultaneously, it extends MzScheme’s units on two points.

1. Linking in compound units is done by name, instead of position.
2. In unit bodies, definitions are separated from initialization expressions, and during linking, they are all put after all definitions.

Extensions should be a good thing. The first one has been introduced in MzScheme, with \textit{signed} units. The second one has not, to our knowledge.

We will not argue here about the first two restrictions, since they would probably be easily overcome.

But let us examine a bit the consequences of the third one. Restricting definitions to values considerably simplifies the semantics of \textit{invoke}, since it becomes a mutually recursive definition (let rec) of values, followed by a unique initialization expression. In contrast, without this restriction, an operational semantics would have to specify the order of evaluation and to feature a more powerful let rec construct for describing this evaluation. As a consequence, the programmer must explicitly evaluate all its definitions before building a unit, with the inconvenient that they cannot be re-exported, since variables are not values. This is probably not too restrictive, since units are first class, but in some cases, it is annoying, as shown by the following example.

\textbf{Example 2}  Assume that we are supposed to write a unit which \textit{prints} ML-like \textit{type} variables: they are represented as records, but are unnamed, and the unit must therefore choose names ‘\texttt{a}, ‘\texttt{b}, etc..., and provide a \texttt{reset} function.

In \textit{OCaml}, this is done with the functor of figure 2.6, with an internal reference, which the \texttt{print} function increments, and the \texttt{reset} function resets. With \texttt{Unit}, it seems that either the reference would have to be defined outside the unit, which might break the abstraction, or a better workaround has to be found. A \textit{possibility} can be sketched as follows

\begin{verbatim}
let PrintTyVar = fn() =>
let x = ref 'a', 'b' in
unit import error eq
  export printTyvar
...
\end{verbatim}
module PrintTVar
    (Base : sig
      val error : string -> 'a
    end)
  (TVar : sig
    type t
    val eq : t -> t -> bool
  end) = struct

  let vars = ref ([] : (TVar.t * string) list)
  let current_name = ref 'a'
  let reset () = current_name := 'a'; vars := []

  let new_name () =
    let c = !current_name in
    let n = Char.code c in
    if n > 122
      then Base.error "cannot print that many type variables. "
    else
      let c' = char_of_int (n + 1) in
      current_name := c';
      "'" ^ (String.make 1 c)

  let string_of_tyvar v =
    try
      List.find (fun (v', name_v') -> TVar.eq v v') !vars
    with
    | Not_found ->
      let s = new_name () in
      vars := (v, s) :: !vars;
      s

  let print_tyvar fmt v =
    Format.printf fmt "%s" (string_of_tyvar v)

end

Figure 2.6: Printing type variables in OCaml
(Let \( x = e_1 \) in \( e_2 \) is syntactic sugar for \((\text{fn } x \Rightarrow e_2) \ e_1.\).) It consists in wrapping the unit inside a function that defines a local reference and returns the unit, which may use the reference, without breaking the abstraction. The restriction of definitions to values seems therefore reasonable, but still a bit ad hoc. It clearly would be preferable to allow any expression as a definition.

In the other sections, we examine the other aspects of units independently of this important drawback, and of the restrictions.

2.3.3 Types

Still in Flatt’s thesis [35], two successive extensions of \texttt{Unit} with types are presented. \texttt{Unit}\textsubscript{c} introduces constructed types in a simplified form. Declarations of the shape

\[
\textbf{type} \; t = x^1, x^1_{\tau_1} | x^2, x^2_{\tau_2} \odot x_m
\]

are allowed, and should be read as the declaration of a type with two constructors \( x^1 : \tau_1 \rightarrow t \) and \( x^2 : \tau_2 \rightarrow t \), two destructors \( x^1_{\tau_1} : t \rightarrow \tau_1 \) and \( x^2_{\tau_2} : t \rightarrow \tau_2 \), and a filter \( x_m \), which takes a value of type \( t \) as argument and returns \texttt{true} if it is of the form \((x^1 \tau)\) and \texttt{false} otherwise.

Types may be exported (as abstract types), thanks to type linking variables \( s \). The programmer may decide to export the constructors and destructors for his type, or not. In contrast with the manifest types [31] = translucent sums [40] systems, there is no mechanism for externally selecting components, so every use of values related to constructed types are in their unit of definition, or in a unit importing them. As a consequence, there is no need for the usual intricate tricks for referring to abstract types: they are syntactically bound by imports. A type system is presented, which is proved sound, except for variant errors: a term such as \( x^1_{\tau}(x^2 \tau) \) is well-typed.

The second extension concerns type abbreviations. The only difficulty is to prevent recursive type definitions, which is done by keeping track of type dependencies in the the unit types, and detecting cycles at composition site. Exporting type abbreviations as manifest types is not possible yet, but it seems to be easy to add.

Subtyping \textbf{Subtyping} There is no subsumption rule in the typed unit calculi (for algorithmic reasons), but subtyping is inlined in the composition and invoke rules. The well-known problem for subtyping extensible records with symmetric concatenation [42] does not cause trouble here. Symmetric concatenation takes the union of two records, provided labels do not clash. The problem basically is the following. The intuitive subtyping relation between extensible records is that a record defining more labels, with finer types, than another record may safely replace it. This intuition is wrong, because a record with more labels than expected may entail label clash during concatenation. Here, the composition operator coerces its argument in one go to the expected type. Indeed, the intermediate layers (see figure 2.5) avoid unexpected label clashes.

A new idiom for modularization \textbf{A new idiom for modularization} Semantically, modules are not nested, and there is no construction for accessing a definition inside a module. Instead, the idea is that the whole program is a let rec definition, followed by initialization expressions, but that units allow to split it into parameterizable fragments, which may be separately distributed and used. This significantly departs from traditional modules, which are more or less assimilated with records. Module evaluation allows to define a record, and after that, other parts of the program have access to its definitions, through the selection operator. Here, idiomatically, the program structure is less hierarchical: in order to use a definition exported by a unit, the programmer has to merge his code with it, to produce a new unit, which stays flat. (More than that, as only values are allowed as definitions, compound units cannot be defined inside units, thus restricting the possibilities of unit nesting.)

As a canonical example, from [35] again, the diamond problem becomes more or less meaningless with units. In ML, assume for instance a Symbol module, used by functors Parser and Lexer,
Figure 2.7: An example diamond problem, with functors

Figure 2.8: An example diamond problem, with units
both of which are later used by the main functor Reader, as in figure 2.7. Type sharing allows to specify that the Symbol module imported by Parser and Lexer has to be the same, in order for the main program to be correct (cf. the type sharing specification “type sym = Lexer.sym”). Moreover, the linking is performed by first building the real lexer and parser, by applying each functor to Symbol, and then applying Reader to the results. With units, there would be units for the symbol, lexer and parser entities, but instead of referring to an imported unit Symbol, the lexer and parser would refer directly to its definitions, as sketched in figure 2.8. Similarly the Reader unit directly imports the type sym and the functions lex and parse. Linking is done by composing the four units together. (It is not really possible in the unit calculi because of the various restrictions, but the idea should be clear.)

This formalism is an interesting aspect of the PLT work, because it reveals that the complexity introduced by the need for exporting data types and use them outside of their initial scope, as in traditional module systems, might be overcome by different design choices, without losing expressive power. Given the depth of this complexity [28], the issue is worth exploring. A drawback of this approach might arise from the lack of a structured name-space. The dot notation [21] has no meaning with units. One could argue that the name space is even more structured with units, since the internal names are irrelevant to the meaning of the unit, especially during composition. Therefore, the programmer may call variables exactly as she wishes according to the context, linking variables making the connections. However, as argued by Szyperski in [74], it is often convenient to feature both static linking, as when using library functions, so hard-wired imports are still useful. From this perspective, it is interesting to notice that the package system of MzScheme remains, even in the presence of units.

2.4 CMS

After the work of Bracha et al., a contemporary work to those of Duggan et al., and Flatt et al. is Ancona’s PhD work on a semantic characterization of mixin modules [3]. He defines mixin modules with the tools of category theory. After this, Ancona and Zucca reformulated and improved this definition in terms of a calculus with an operational semantics, called CMS [5]. At the same time, Wells and Verstergaard developed their m-calculus [76], which is similar to CMS in many ways. These two contributions are less pragmatic and more foundational than previous work on mixin modules. We give an overview of both of them, and compare their respective merits, beginning with CMS in this section. For both calculi, we change notations and names a bit for homogeneity reasons.

2.4.1 Syntax and semantics

Syntax CMS [5, 6] defined by the pseudo-syntax in figure 2.9. Contrarily to Flatt and Felleisen’s work, the distinction between a-convertible variables and fixed external names is here syntactically enforced: variables are ranged over by $x, y, z$ and names are ranged over by $X, Y, Z$.

CMS is parameterized over an arbitrary core language, with some conditions, not explicited here, since they are very intuitive, see [5] for details. $C$ denotes a core expression. In CMS, a core expression must be wrapped in an explicit substitution $\rho$ in the style of [2], which must cover the whole set of its free variables, and not be recursive ($\text{FV}(\text{cod}(\rho)) \perp \text{dom}(\rho)$).

CMS basic modules are constructed by the $[; \sigma; \rho]$ form. The meta-variable $\iota$ ranges over input assignments, which are lists of bindings from variables to names, written $x_i \mapsto X_i$. The notation is used also below for output and local assignments, in the same sense. Assignments must correspond to surjective finite maps and the $x_i$s must be different. Output assignments $\rho$ map names to expressions, and represent the definitions exported by the module. Local assignments $\rho$ map variables to expressions, and are the hidden definitions of the module. The scope of the variables
Expression: \[ E ::= x \]
\[ C[\rho] \]
\[ [i; o; \rho] \]
\[ E_1 + E_2 \]
\[ \sigma_1[E] \]
\[ freeze_\eta(E) \]
\[ E.X \]

Finite maps:
\[ \iota ::= x_i \in \mathcal{I} X_i \]
\[ \sigma ::= X_i \in \mathcal{I} E_i \]
\[ \rho ::= x_i \in \mathcal{I} E_i \]
\[ \sigma_i ::= X_i \in \mathcal{I} Y_i, Y_j \in \mathcal{J} \]

**Figure 2.9: CMS syntax**

bound by \( \iota \) and \( \rho \) is the whole mixin module. A basic module is well-formed if \( \iota \) and \( \rho \) do not bind any variable in common. Composition \( \sigma_1 \circ \sigma_2 \) is defined on finite maps, only if \( \text{cod}(\sigma_2) \subseteq \text{dom}(\sigma_1) \).

Union \( \sigma_1 + \sigma_2 \) is defined on finite maps \( \sigma_1 \) and \( \sigma_2 \), provided \( \text{dom}(\sigma_1) \cup \text{dom}(\sigma_2) \).

Module operators include composition, here called the sum, which links two modules together. The reduct operator \( \sigma_1[E] \), is roughly a powerful renaming operator, but not only, since it expresses definition hiding. Here \( \sigma' \) and \( \sigma'' \) are renaming, which syntactically are pairs of assignment mapping names to names and a list of names, which we call the unused names. The unused names must not be in the codomain of the assignment. In other terms, renamings are finite maps, as assignments, but they are not forced to be surjective. CMS also includes a powerful freezing operator, for making some definitions early bound, and the usual selection operator.

Variables are \( \alpha \)-convertible in basic modules, and we will consider expressions modulo \( \alpha \)-conversion.

**Semantics** The semantics of CMS is defined as the least contextually closed relation respecting the rules in figure 2.10. The rules only apply when both sides of the \( \rightarrow \) symbol are well-formed expressions. The strength of CMS is the way inputs and outputs are kept separated, which allows for very powerful yet simple operators.

By rule **AZ-Core**, the reduction relation \( \rightarrow \) of CMS includes the transitive closure of the reduction on core expressions \( \rightarrow_{\text{C}} \), which is a parameter of the system. Moreover, sometimes the evaluation of a core expression \( C[\rho] \) can require the value of a variable \( x \), explicitly bound in the surrounding substitution \( \rho \). Then, by rule **AZ-Sub**, if the expression to which \( x \) is bound has the form \( C'[\rho_1] \), then \( x \) is replaced with \( C' \) in \( C \), (thanks to the core substitution, which is also a parameter of the system,) while the pending substitution now includes the bindings in \( \rho_1 \), and no longer binds \( x \).

Rule **AZ-Sum** simply takes the unions of the present finite maps, provided no clash or variable capture occurs. Specifically, writing \( \alpha_1 + \alpha_2 \) implies that \( \alpha_1 \) and \( \alpha_2 \) have disjoint domains. Inputs are shared, but variables mapping to the same name are kept different. Another operator, called the left preferential sum \( E_1 \leftarrow E_2 \) is also defined, which does not require the outputs to be disjoints, but rather gives precedence to outputs coming from the right. It is defined by the following reduction rule:

\[
\frac{(\text{dom}(\iota_1) \cup \text{dom}(\rho_1)) \perp (\text{FV}([t_2; o_2; \rho_2]))}{(\text{dom}(\iota_2) \cup \text{dom}(\rho_2)) \perp (\text{FV}([t_1; o_1 + o; \rho_1]))} \quad \frac{\text{dom}(\sigma) \subseteq \text{dom}(\rho_2)}{[t_1; o_1 + o; \rho_1] \leftarrow [t_2; o_2; \rho_2] \rightarrow [t_1 + t_2; o_1 + o_2; \rho_1 + \rho_2]} \quad \text{AZ-Override}
\]
Rule AZ-REDUCT describes the action of the reduct operator $\rho$, $E_{|\rho|}$. In fact, it could be divided into two operators, one for reducing input $\rho$, $E$, and one for reducing outputs $E_{|\rho|}$. Both actions are similar though. Each of them bases on a renaming $\sigma^i$ and $\sigma^o$, respectively. Input renaming changes the input names, but not the corresponding variables, and possibly adds new (unused) input variables. This is done by composing the renaming with the former input assignment. For instance, if a name $X$ is renamed into another name $Y$, then $\sigma^i$ includes a binding $X \mapsto Y$, and $\sigma^o$ includes a binding $x \mapsto X$. Then, the composition of $\sigma^i$ and $\sigma^o$ has a binding $x \mapsto Y$. If the renaming has unused names, or if $dom(\sigma^i)$ contains names not in $cod(\sigma^i)$, then a fresh variable is associated to each of them, thus adding dummy inputs. If the renaming is not injective, then some inputs get shared. Similarly, output renaming composes the renaming with the output assignment, possibly forgetting some exported names. Renaming $X$ to $Y$ as above would here be done with a binding $Y \mapsto X$, the initial output assignment having a binding $X \mapsto E$. The renamed output, $o \circ \sigma^o$, then has a binding $Y \mapsto E$.

The freeze operator, described by rule AZ-FREEZE, makes some definitions early bound in a mixin module. As an additional argument, it takes a renaming from some input names to some output names. The finite map tells which definitions must be associated to the frozen input names. The internal variables corresponding to the frozen names are definitely bound as local definitions. As an example, consider the following mixin module

$$
\text{freeze}_{X \mapsto Z, Y \mapsto Z}([X \mapsto x, Y \mapsto y; Z \mapsto E;])
$$

which reduces to

$$
[; Z \mapsto E; x \mapsto E, y \mapsto E].
$$

As a side observation, CMS does not at all bother with sharing computations.

Finally, a mixin module without any input is ready to be used by the outer world.

**Definition 3 (Concrete and open mixin modules)** A mixin module is said concrete if it does not have any input. Otherwise, it is called open.

Rule AZ-SELECT selects a definition out of a concrete mixin module $[\sigma; \rho]$, with $\rho = x_i \mapsto I E_i$. One cannot simply copy the body $E = o(X)$, because it might contain references to the local definitions. Such internal calls are implemented as follows. Each free occurrence of $x_i \in dom(\rho)$ in $E$ is replaced with a kind of closure: the local definitions $\rho$ of the mixin module are put in a new mixin module, which only exports a name $Y$, bound to the definition $E_i$. $x_i$ then corresponds to selecting $Y$ in this mixin module.

The operational semantics of CMS is confluent, as stated by the following theorem by Ancona and Zucca [5].

**Theorem 1 (CMS is Church-Rosser)** If $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$, then there exists $E'$ such that $E_1 \rightarrow^* E'$ and $E_2 \rightarrow^* E'$.

### 2.4.2 Types

CMS is equipped with a type system that reflects the distinction between input and outputs. Core expressions have core types, and mixin modules have types of the form $[\Sigma^i; \Sigma^o]$, where $\Sigma^i$ and $\Sigma^o$ are signatures, i.e. finite sets of pairs of a name and a type. $\Sigma^i$ is the input signature, representing the requirements put on inputs, while $\Sigma^o$ is the output signature, declaring the capabilities offered by the output definitions. Typing judgments are parameterized by the corresponding judgments.
\[
\begin{align*}
    C \rightarrow^{+} C' & \quad \text{(AZ-CORE)} \quad C[x \mapsto C'[ρ_1, ρ_2]] \rightarrow C[x \mapsto C'][ρ_1, ρ_2] \quad \text{(AZ-SUB)} \\
    C[ρ] \rightarrow C'[ρ] & \\
    E_1 = [t_1; o_1; ρ_1] \quad E_2 = [t_2; o_2; ρ_2] \quad BV(E_1) \perp FV(E_2) \quad BV(E_2) \perp FV(E_1) \quad \text{(AZ-SUM)} \\
    E_1 + E_2 \rightarrow [t_1 + t_2; o_1 + o_2; ρ_1 + ρ_2] \\
    \sigma^* [t; o; ρ] \rightarrow [σ^*(t; o; ρ)] \quad \text{(AZ-REDUCE)} \\
    \text{cod}(t_2) \perp \text{dom}(σ^f) \quad \text{(AZ-FREEZE)} \\
    [; o; x_i \mapsto i \in I E_i], X \rightarrow o(X)[x_j \mapsto j \in J \rightarrow i \in I E_i], Y \quad \text{(AZ-SELECT)}
\end{align*}
\]

Figure 2.10: Reduction rules for CMS

on core expressions and types. The typing rules are presented in figure 2.11. Our presentation is a bit different from that of Ancona and Zucca, in that we do not use type annotations to guide a possible typing algorithm. In a sense, our presentation could be viewed as the Curry-style version of their Church-style presentation.

By rule AZ-T-VAR, a variable has the type it is assigned in the environment. By rule AZ-T-CORE, the explicit substitution construct is typed as a let binding. The bound expressions \(E_i\) must have core types. They are added to the environment to type the final core expression \(C\), thanks to the core type system.

Rule AZ-T-BASIC describes the typing of a basic mixin module. A type has to be guessed for each bound variable, those of the input \(i = x_i \in I X_i\) and those of the local definitions \(ρ = x_k \mathrel{\forall K} E_k\), say \(x_i : τ_i \in I \cup K\). With these types, the local definitions can be checked to have the expected types \(τ_k \in K\), and the exported definitions can be typed \(τ_j \in J\). The final type of the expression can then be formed: it has the input types as an input signature, and the export types as an output signature. This type must be checked well formed, which means that the signatures are finite maps.

Rule AZ-T-SUM describes how the sum of two mixin modules is typed. Provided the two outputs do not define any name in common, the sum takes the union of the input types and of the output types. Thus, some common inputs can be shared during composition. The result type must be checked well formed, as for instance two similarly named inputs could have different types in the two input signatures.

Rule AZ-T-REDUCT, given that the argument mixin module has type \([Σ', Σ''\] \], guesses two signatures \(Σ'\) and \(Σ''\) such that the input and output renamings respectively map \(Σ'\) to \(Σ'\) and \(Σ''\) to \(Σ''\), preserving types, as witnessed by the side-conditions \(σ^' : Σ' \rightarrow Σ'\) and \(σ^'' : Σ'' \rightarrow Σ''\). Notice that the renamings are allowed not to be surjective, which lets some choice to the type system in attributing types to the names that are not present in the original type.

Similarly, rule AZ-T-FREEZE checks that the freezing map \(σ^f\) maps some input specifications to output specification, preserves types, and removes the frozen declarations from the input signature.

Finally, in the case of a selection, rule PROJECT choose the type associated to the selected name.
2.4.3 Expressiveness and inconveniences

**Encodings** In [6], Ancona and Zucca present encodings for the untyped λ-calculus, which accounts for an encoding of ML-style module systems, an encoding of Abadi and Cardelli’s ACC calculus of objects [1]. This accounts for the computational power of the calculus. We informally present the two encodings.

The λ-calculus is easily encoded by using the abstraction facility provided by mixin modules: choose two reserved names ARG and RES, and encode any function λ.x.e, as [x → ARG; RES → e]; (where [ ] denotes the encoding function). Function application e₁e₂ can then be expressed as

\[(freeze_{ARG}→ARG([e₁] + [; ARG → [e₂];]));RES.\]

During application, the ARG field is filled with the translation of the argument, and then frozen. Computation is then triggered by selection of the RES field from the result. The usual α conversion and β reduction are modeled by this encoding.

Encoding Abadi and Cardelli’s ACC calculus of objects is more difficult, so we do not detail it here. Basically, the SELF parameter is modeled as a deferred component, and each method is defined as an output component. Overriding is trivial to implement with the left preferential sum of section 2.4.1. For method calls, the SELF input has to be filled with a definition. The adopted solution, introduced in [1] already, consists in filling it with the object itself. The result can then be frozen and the needed method selected. A method call e.I is then encoded as \(freeze_{SELF→SELF([e] + [; SELF → [I];])}.I\).

**CMS as an implementation language and other operators** Also, CMS is further used as an implementation language for DCMS, a typed surface language with mixin modules. The interest
is that the power of CMS is used to encode more usual operators. The fundamental difference between CMS and DCMS is that mixin module types are divided into deferred, virtual, and frozen components. Virtual components correspond to CMS components present both as inputs and as outputs, whereas frozen components are those that are only outputs. This allows for a refined overriding policy: frozen components cannot be deleted in DCMS, whereas virtual ones can. This policy is only enforced by typing, since the dynamic semantics of DCMS is given by translation to CMS.

Additionally, the operators are not exactly the same as in CMS. For instance, the reduct operator is split into more atomic operators. For instance, restriction allows to delete some virtual definitions, and freezing only allows to map input components to virtual components of the same name. In fact, renaming is not possible at all anymore. We think that it should have been maintained, maybe as a separate operator. Hiding takes some virtual and frozen components, freezes the not already frozen ones, and deletes them from the result. Thus, other definitions will continue using the hidden definitions even if at some point the corresponding names are defined again, differently. Finally, selection allows to select a component from a mixin module that still has some virtual components (which semantically corresponds to first freezing those components before to perform the selection).

The type system of DCMS does not guarantee that frozen definitions of a mixin module will remain the same whatever use can be made of it, as is the case for final class methods in Java [48]. It does not seem too far from it though: probably, only the hiding operator breaks this property. According to the authors [1], frozen components are closer to static methods in Java than to final methods. However, it should not be difficult to encode final methods with a refined typing policy.

**CMS is call-by-name** In call-by-name or lazy programming languages such as Haskell [9], modules are basically finite sets of definitions, i.e. unevaluated code. In that sense, CMS concrete mixin modules are rather similar to modules. The operational semantics given by Ancona and Zucca does not model the sharing taking place with the lazy strategy for instance, but it is rather a matter of level of abstraction or of presentation than a semantic inadequacy. Moreover, computational aspects of modules, especially with respect to monads, can be easily introduced in CMS, as shown by Ancona et al. [4].

On the contrary, in call-by-value languages, a module is rather a piece of code at first, which is evaluated, and results in a set of values. The definitions contained by a module are thus evaluated prior to be used by other parts of the program. As an example, consider the very simple module (in OCaml-style syntax)

```ocaml
struct
  let f x = x + 1
  let res = f 0
end
```

Intuitively, this module should be represented in CMS as \([f \mapsto F, res \mapsto RES; F \mapsto \lambda x.x + 1, RES \mapsto f0;] \). However, there is no hope that this mixin module reduces to the expected value, i.e. \([f \mapsto F, res \mapsto RES; F \mapsto \lambda x.x + 1, RES \mapsto 1;] \). Indeed, in CMS modules, one definition is never allowed to use definitions of the same mixin module. And this is coherent with the late binding semantics: if we override \(F\) above, the definition of \(RES\) must use the new definition. As a consequence, in our quest for a call-by-value language with mixin modules, we need a mechanism for triggering module evaluation, reminiscent of Duggan and Sourleis’ \(\texttt{close}\) operator, and of Flatt’s instantiation operator. In the following, we call this operation \texttt{close}. Mixin modules have to contain unevaluated code, because of the late binding semantics, but they must be mapped somehow to the usual call-by-value notion of modules, by evaluating their definitions.

---

Hint 1 (Close operator) In a call-by-value setting, mixin modules should contain unevaluated code, and the language must feature a close operator for triggering this code, thereby transforming concrete any mixin module into a module, the definitions of which can then be selected by the rest of the program.

Types, recursion, and call-by-value In call-by-value languages, it is usual to restrict recursive definitions to syntactic functions [58], possibly with some extensions [55]. Such restrictions rule out some ill-founded recursive definitions, and that they allow more efficient compilation.

Nevertheless, with respect to recursive definitions, mixin modules go farther than conservativity. Indeed, arbitrary recursive definitions can appear at runtime, as we saw in section 1.2.4. It is undesirable that mixin modules force language designers to restart writing their compilers from scratch, or to forget about their useful optimizations.

Therefore, it is important to find a way of statically ruling out forbidden recursive definitions, which the type system of CMS does not provide.

2.5 The m-calculus

In [76, 75], Wells and Vestergaard present the m-calculus. It is presented as a calculus for linking, but according to definition 2, it features mixin modules. We describe it briefly in this section.

2.5.1 Syntax and semantics

Syntax The syntax of the m-calculus is presented in figure 2.12. Syntactic conventions are similar to those chosen for CMS: variables \(x, y, z \in \text{Vars}\) are distinguished from names \(X, Y, Z \in \text{Names}\). A mixin module \(\langle b \rangle\) in m consists of binding \(b\). A binding is a list \(\langle d_1 \ldots d_n \rangle\) of definitions \(d_i\). A definition \(d = \langle L \triangleright x = B \rangle\) binds a label \(L\) and a variable \(x\) to a definition body \(B\). A label is either a name \(X\) or the anonymous label \(\_\) which allows to write local definitions. A body is either an expression \(e\) or the empty body \(\bullet\), which allows to write input definitions. An expression can be a variable \(x\), a module \(\langle b \rangle\), the linking of two expressions \(e \triangleright f\), the hiding of a name in an expression \(e.\_\), the selection of a name in an expression \(e.X\), and the mutually recursive binding of expressions in another one, let rec \(b\) in \(e\).
Syntactic correctness  Some conditions are required for syntactic correctness.

- First, bindings should not bind the same name or variable twice.
- Second, bindings in let rec should be anonymous and non-empty, i.e. of the shape \( \triangleright \ x = e \).
- There are no unnamed, empty definitions.

Contrarily to CMS, input, output, and local definitions are here mixed in the same binding. We recover the same structure though: inputs are named, empty definitions, outputs are named, non-empty definitions, and locals are unnamed, non-empty definitions. Bindings can be seen as finite maps from pairs of a label and a variable to bodies. We will use standard operations on finite maps on them, such as the union \( + \). By slight abuse of notation, we denote by \( b|_{\mathcal{N}} \) (where \( \mathcal{N} \) is a set of names) the restriction of the finite map \( b \) to definitions named with an element of \( \mathcal{N} \). Similarly, we denote by \( b|_{\mathcal{N}} \) the restriction of the finite map \( b \) to anonymous definitions and definitions named with an element out of \( \mathcal{N} \). We do the same abuse of notation for variables, in particular, for designing the definition associated to a variable \( x \) in a binding \( b \), we write \( b(x) \).

Structural equivalence  Variables in mixin modules and let rec are \( \alpha \)-convertible, as usual. Moreover, expressions are considered equivalent modulo commutation of the arguments to a linking, and modulo the order of definitions in a binding.

Dynamic semantics  The dynamic semantics of \( m \) is defined as the least contextually closed relation respecting the rules in figures 2.13.

The main and most complicated rule is the WV-LINK for linking two mixin modules \( (b) \) and \( (b') \).

First, the definitions bound by the same names in the two bindings are isolated, the other ones being copied straightforwardly into the result mixin module. The notation \( DN(b) \) denotes the set of names defined by a binding, so \( \mathcal{N} = DN(b) \cap DN(b') \) is the set of names defined in common by the two mixin modules. Let those common definitions be \( b|_{\mathcal{N}} = (X_1 \triangleright x_1 = B_1 \ldots X_n \triangleright x_n = B_n) \) and \( b'|_{\mathcal{N}} = (X_1 \triangleright x_1 = B'_1 \ldots X_n \triangleright x_n = B'_n) \). Then, for each pair of similarly named definitions, the function \( PickBody \) choose the non-empty body if any, and otherwise denotes \( \bullet \). \( PickBody(\bullet, e) = PickBody(\bullet, e) = e \) and \( PickBody(\bullet, \bullet) = \bullet \). In the case of two non-empty bodies, \( PickBody \) is undefined, and thus if the rule applies, it implies that no such conflict occurs. Notice that contrarily to CMS, variables binding the same names in the two bindings are assumed to be equal here. It can be reached by structural equivalence, of course, exactly as the condition imposed by CMS.

Rule WV-ISUBST (for internal substitution) describes the use of a definition to evaluate another definition in the same binding. If a definition is of the shape \( L_1 \triangleright x_1 = C[x_2] \), and \( x_2 \) binds another definition \( L_2 \triangleright x_2 = e \), it is allowed to copy \( e \) into \( C[x_2] \), provided no capture occurs and the second definition does not risk to depend on the first one. This is formalized by considering the dependency graph \( \rightarrow_{(b)} \) of our binding \( b \). This graph has the variables defined by the binding as nodes, and its edges are build as follows. If the body of a definition \( L_1 \triangleright x_1 = e_1 \) is non-empty and has \( x_2 \) as a free variable, then there is an edge \( x_2 \rightarrow_{(b)} x_1 \). Moreover, an input definition (a definition with an empty body) potentially depends on all the named definitions of the binding, so there are edges from each of the variables binding named definitions to all empty definitions. We say that a definition \( d_1 \) depends on a definition \( d_2 \) if the reflexive, transitive closure of the dependency graph has and edge from the variable binding \( d_2 \) to the one binding \( d_1 \). To sum up, rule WV-ISUBST allows substitution outside of dependency cycles. The reason for this restriction is that confluence would be lost otherwise, as noticed by Ariola and Klop in [8]. Notice that rule WV-ISUBST applies as well in mixin modules as in let rec.

In let rec, however, the values defined in the binding can also be used in the body of the let rec, as stated by rule WV-ESUBST. The side-condition just ensures that no variable capture occurs, and that the occurrence of \( x \) in the body of the let rec actually refers to the considered binding of \( x \).
\[ N = \text{DN}(b) \cap \text{DN}(b') \quad b_{\downarrow N} = (X_1 \triangleright x_1 = B_1 \ldots X_n \triangleright x_n = B_n) \]
\[ b'_{\downarrow N} = (X_1 \triangleright x_1 = B'_1 \ldots X_n \triangleright x_n = B'_n) \quad \forall 1 \leq i \leq n, B'_i = \text{PickBody}(B_i, B'_i) \]
\[ \langle b \rangle \oplus \langle b' \rangle \rightarrow \langle b_{\downarrow N} + b_{\downarrow N} + b'' \rangle \]  
(WV-LINK)

\[ b = (L_1 \triangleright x_1 = C[x_2], L_2 \triangleright x_2 = e, b') \]
\[ \text{Capt}_{\cap}(C) \perp \{x_2\} \cup \text{FV}(e) \quad x_1 \not\in_{(x_2)} x_2 \]
\[ b \rightarrow (L_1 \triangleright x_1 = C[e], L_2 \triangleright x_2 = e, b') \]  
(WV-SUBST)

\[ \text{let rec } b \text{ in } C[x] \rightarrow \text{let rec } b \text{ in } C[b(x)] \]  
(WV-ESUBST)

\[ \text{DN}(b') = \emptyset \quad \text{DV}(b') \perp \text{FV}(b) \quad b' \neq e \]  
(WV-GC-MODULE)

\[ \text{DV}(b') \perp (\text{FV}(b) \cup \text{FV}(e)) \quad b' \neq e \]
\[ \text{let rec } b + b' \text{ in } e \rightarrow \text{let rec } b \text{ in } e \]  
(WV-GC-LETREC)

\[ \text{let rec } e \text{ in } e \rightarrow e \]  
(WV-EMPTY-LETREC)

\[ b' \neq e \]
\[ \text{let rec } b' \text{ in } \langle b \rangle \rightarrow \langle b' + b \rangle \]  
(WV-CLOSURE)
\[ X \notin \text{DN}(b) \]
\[ \langle b \rangle; \ x \rightarrow \langle b \rangle \]  
(WV-HIDE-ABSENT)

\[ \langle x \triangleright x = B + b \rangle; \ x \rightarrow \langle x \triangleright x = B + b \rangle \]  
(WV-HIDE-PRESENT)

**Figure 2.13: Reduction rules for m**

The WV-GC-MODULE rule describes the garbage collection of a non-empty set of unused local definitions, and similarly, the WV-GC-LETREC rule describes the garbage collection of a non-empty set of unused definitions in a let rec.

The rule WV-CLOSURE describes the elimination of let recs. What happens when an argument of a linking operation turns out to evaluate to an expression of the shape let rec \( b \) in \( \langle b' \rangle \)? The rule WV-LINK does not apply directly. Contrarily to Ariola et al. [8, 7], who lift the let recs to the top of the expression, Wells and Vestergaard choose to merge the let recs into the mixin module, as formalized by the WV-CLOSURE rule. The expression above reduces to \( \langle b + b' \rangle \). This treatment of let rec resembles explicit substitutions [2], and is possible because all strict operators expect mixin modules as arguments.

Finally, rules WV-HIDE-ABSENT and WV-HIDE-PRESENT define the semantics of definition hiding. By rule WV-HIDE-ABSENT, hiding an absent definition does nothing, whereas by rule WV-HIDE-PRESENT, hiding a present definition replaces its name with the anonymous label.

**Properties of the reduction relation** The reduction relation is confluent, and enjoys the strong finite developments property [75]. Roughly, this means that reducing all the redexes present in an expression and their residuals in any order leads to a unique normal form.

47
2.5.2 Expressiveness and inconvenient

Wells and Vestergaard [75] show encodings for even more features than Ancona and Zucca. Besides record operations, first-class functions, and Abadi and Cardelli’s object calculus (ACC), Wells and Vestergaard show encodings for C-style modules, packages (Haskell style), higher-order ML style modules, at least of their type-free aspects. Finally, they compare the expressiveness of $m$ with other calculi for linking, including first-class contexts [44], first-class environments [68, 67], and CMS. An encoding of CMS is given, which is not exactly a simulation, but is conjectured to preserve observable behaviour. This encoding is interesting because $m$ initially features neither definition renaming nor late binding, and the encoding shows that they are in fact present in the calculus, in a quite intuitive way.

**Late binding** The set of names can be partitioned into input names - written here with the superscript $i$, as in $X^i$ - and output names - written here with the superscript $o$.

A virtual definition of CMS named $X$, i.e. a couple of an input $x \mapsto X$ and an output $X \mapsto e$ can then be represented by a couple of an input definition $X^i \mapsto x = \bullet$ and an output definition $X^o \mapsto y = e$. The variable $y$ must not be used by any definition. This way, the WV-ISUBST never applies for virtual definitions, thus preserving the late binding semantics. Overriding can then be implemented by first hiding the definition of $X^o$, then garbage-collecting it (since $y$ is unused), and finally linking with a mixin module defining $X^o$ again.

**Definition renaming** A *positive* atomic renaming in $m$ is a pair of names, written $X^+ := Y$. It is used for renaming output definitions, and applied to an expression $e$ by

$$(e \oplus \langle X \mapsto x = \bullet, Y \mapsto y = e \rangle) : x.$$  

The effect is that the definition provided by $e$ is bound to $x$ and re-exported as $Y$. When $X$ is then hidden, $Y$ still exports the right definition, and has semantically replaced $X$ in the interface of the mixin module.

A *negative* atomic renaming $X := Y$ allows to rename input definitions, by a dual mechanism. It is applied to an expression $e$ by applying the inverse positive atomic renaming, i.e. $X^- := Y$.

A more complicated notion of simultaneous renaming is given, which follows the same idea, but is slightly more powerful since it allows to duplicate output components and to merge input components.

Additionally (atomic) renaming has an action that recalls freezing. If an input name is renamed to an output name, this has the effect of resolving the input with the output, as shown by the following example reduction.

$$
\begin{align*}
\langle X \mapsto x = \bullet, Y \mapsto y = e \rangle &\mid X := Y \\
= &\ (\langle X \mapsto x = \bullet, Y \mapsto y = e \rangle \oplus \langle Y \mapsto y = \cdot, X \mapsto x = y \rangle) : x \\
\rightarrow &\ \langle X \mapsto x = y, Y \mapsto y = e \rangle : x \\
\rightarrow &\ \langle X \mapsto x = y, Y \mapsto y = e \rangle
\end{align*}
$$

The result mixin module is observationally equivalent to $\langle X \mapsto x = e, Y \mapsto y = e \rangle$, which corresponds to the result of freezing $X$ as $Y$.

Thus, $m$ and CMS offer similar features, from the standpoint of dynamic semantics. It is interesting to list the differences, and record what they bring to the theoretical study of mixin modules.
2.5.3 Comparison with CMS

Shape of the basic mixin modules and subtyping  Basic mixin modules in the m-calculus are very close to an acceptable concrete syntax, because deferred, local, and exported definitions can be interleaved. Furthermore, this gives an intuition about a possible order of evaluation of the definitions. However, the shape of CMS basic mixin modules provides more information about the status of definitions. In particular, in CMS, a definition can be exported without being imported (i.e. without being bound by a variable). In the m-calculus, this information is hidden in the fact that the variable binding the observed definition is unused.

This has consequences on typing, especially in the presence of depth subtyping, which will probably appear more natural in CMS. Depth subtyping is likely to work well with mixin modules, because the inputs are contravariant, whereas the outputs are covariant, and the distinction appears in types. (Width subtyping for mixin modules has to do with extensible records subtyping [42, 63, 11], and is not concerned here.) With CMS-like basic modules, it is intuitive to specify the import type of a virtual component $X$, even if it is different from the type of the definition. For instance, a typed mixin module like $[x \to X : T; Y \to e]$ would have a type like $[X : T; X : T']$, where $T'$ is the type of $e$. The constraint is that $T'$ must be a subtype of $T$. It is then possible to delete $X$, and replace it with a definition of type $T''$, as long as it is also a subtype of $T$. In the m-calculus, the shape of basic mixin modules suggests that an exported definition will only be specified by one type, which compromises this kind of feature. Of course, this is just syntax, and a distinction on input and output can be made for types, even if it does not appear in expressions.

We conjecture that this form of subtyping allows for a great simplification of a paper by Bono et al. on subtyping mixins in a mobile setting [10]. Indeed, in their framework, the subtyping points are clearly located at receive time, which allows for automatic coercion insertion, with width subtyping. More precisely, for width subtyping, a mixin module $E$ of type $\Sigma^i; \Sigma^e$ can be coerced to the type $\Sigma'^i; \Sigma'^e$, provided the following inclusions hold: $\Sigma^i \subset \Sigma'^i$, $\Sigma'^e \subset \Sigma^e$. The coercion is $\tau[E]_{\sigma^i}$, where $\sigma^i$ and $\sigma^e$ are the canonical injections from $\text{dom}(\Sigma^i)$ to $\text{dom}(\Sigma'^i)$ and from $\text{dom}(\Sigma'^e)$ to $\text{dom}(\Sigma^e)$, respectively.

Independence with respect to the core language  The most obvious semantic difference between both calculi is that CMS features second-class mixin modules, explicitly abstracting over an almost arbitrary core language. The standpoint of m is rather to have first-class mixin modules, and rely on the expressiveness of the mixin module language to account for other core language features.

Partly because of the parameterization over the core language, CMS seems more difficult to adapt to a call-by-value setting. In CMS, when the evaluation of a core expression uses a recursive definition, it can be represented by selecting a component out of a mixin module, and storing it in a closure. Consider the following example, where the core language is assumed to include integers, arithmetic operators, and an if ... then ... else ... operator. Let $C =_{\text{def}} \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } \text{fact}(x - 1)$, and $E_0 =_{\text{def}} C[\text{fact} \mapsto \text{fact}]$, which injects $C$ into the mixin module language. Then, the mixin module $E_1 =_{\text{def}} [,\text{FACT} \mapsto E_0; \text{fact} \mapsto E_0]$ exports a function FACT that computes the factorial of an integer argument. Suppose now that we want to compute the factorial of 0.

Here is how the reduction proceeds

$$(\text{fact } 0)[\text{fact} \mapsto E_1.\text{FACT}]$$

$\rightarrow (\text{fact } 0)[\text{fact} \mapsto (E_0 [\text{fact} \mapsto E_1.\text{FACT}])]$ (by rule AZ-SELECT)

$\equiv (\text{fact } 0)[\text{fact} \mapsto (C[\text{fact} \mapsto E_1.\text{FACT}])]$

$\rightarrow (\text{fact } 0)[\text{fact} \mapsto C][\text{fact} \mapsto E_1.\text{FACT}]$ (by rule AZ-SUB)

$\equiv (C 0)[\text{fact} \mapsto E_1.\text{FACT}]$

$\rightarrow 1[\text{fact} \mapsto E_1.\text{FACT}]$
In CMS, the obtained expression is not a value and loops, trying to evaluate the closure. However, one could imagine a garbage collection rule for unused bindings in closures. Unfortunately, in a call-by-value setting, the reduction would evaluate the \( E_i.FACT \) in the closure first, and unfortunately this would not terminate, since the same expression appears again after one reduction step. In this thesis, we will choose the \( \textbf{m} \) way, and rely on the mixin module language to express usual core language features.

**From concrete mixin modules to modules** In \( \textbf{m} \), a mixin module without input definitions and without dependency problems can be evaluated, thanks to the WV-ISUBST rule. What we call a dependency problem is a case where one or more definitions would need to copy the value of each other in order to evaluate. For instance, an expression such as \( X > x = x \odot x \) has a dependency problem. Nevertheless, all the common recursive definitions behave well in the \( \textbf{m} \)-calculus. A recursive definition of functions, for instance, is perfectly evaluated and can be further used by other definitions, in the same binding. In this respect, \( \textbf{m} \) is not as call-by-name as CMS. In CMS, a concrete mixin module (see definition 3) with unevaluated definitions never further evaluates. This fact leads to the conclusion that \( \textbf{m} \) is close to a language of call-by-value mixin modules.

Indeed, if mixin modules are represented as explained by the encoding of section 2.5.2, we have seen that the late binding semantics is preserved. Call-by-value mixin modules could then be defined by restricting the reduction relation to a call-by-value strategy. Roughly, this could be done as follows.

- Refine the notion of value. Open mixin modules are values (do not evaluate inside open mixin modules). Only fully evaluated concrete mixin modules are values.
- Restrict both substitution rules WV-ISUBST and WV-ESUBST to copy only values,
- Restrict selection to value concrete mixin modules.
- (Maybe this could also require to modify the handling of let rec bindings.)

The obtained calculus respects hint 1, by distinguishing open mixin modules from concrete ones, only allowing evaluation inside the latter, and selecting components only from evaluated concrete mixin modules. The close operator is not directly in the language, but its role can be played by freezing. If we call virtually concrete mixin modules the ones such that all input names correspond to an output name, closing a virtually concrete mixin module can be done by freezing all its components.

The obtained language still remains unsatisfactory though, in at least two aspects.

- First, there is a need for a polymorphic close operator. Indeed, the above solution only encodes close locally, for a mixin module whose shape is known.
- Second, once a virtually concrete mixin module has been closed (by freezing all its input components), evaluation remains undeterministic (as the evaluation of let rec is).

The first point is not too hard to solve: closing a virtually concrete mixin module consists in replacing the input variables with the corresponding output variables, and removing the input declarations, thus making the mixin module concrete and ready for evaluation.

The second point is more problematic however. Making evaluation deterministic turns out difficult. Indeed, in \( \textbf{m} \), bindings are considered equivalent modulo reordering of definitions. But given a binding, evaluation has to find a unique correct order of evaluation for definitions, that does not violate dependencies. The uniqueness comes from the requirement that in a call-by-value language, side-effects must appear in a predictable order. In usual call-by-value module systems, the order of
evaluation is given syntactically, but here, in some cases, it does not exist. For instance, consider the binding \( _\triangleright x = 1 + 2, _\triangleright y = 2 \times 1 \). There are two possible orders of evaluation.

A first idea to solve the problem is to stop considering bindings equivalent modulo the order of definitions, and specify an order of evaluation inside them, say from left to right. However, this breaks the definition of linking. As an example, consider the two mixin modules \( e_1 \equiv (X \triangleright x = \cdot, Y \triangleright y = x + 1) \) and \( e_2 \equiv (X \triangleright x = 0) \). According to the semantics of \( m \), \( e_1 \otimes e_2 \) can be either \( (X \triangleright x = 0, Y \triangleright y = x + 1) \) or \( (Y \triangleright y = x + 1, X \triangleright x = 0) \), alternatively. Assume that we define linking to remove empty definitions when a non-empty one is provided, so that non-empty definitions do not change their relative positions. (A semantics remains to be given for the case where two empty definitions meet, but this informal discussion does not specify it.) The above linking then results in \( (Y \triangleright y = x + 1, X \triangleright x = 0) \), whose evaluation fails, because the value of \( x \) is needed to evaluate \( x + 1 \), and \( X \triangleright x = 0 \) is to the right of \( Y \triangleright y = x + 1 \).

In many cases, this will appear too rigid.
Part II

Dynamic and static semantics of call-by-value mixin modules
Chapter 3

Dynamic semantics: the MM language

3.1 Syntax

The syntax of MM is defined in figure 3.1. The meta-variables $X$ and $x$ range over names and variables, respectively. Variables are used as binders, as usual. Names are used for accessing to definitions in mixin modules, as an external interface to other parts of the expression. Figure 3.2 recapitulates the meta-variables and notations we introduce in the remainder of this section.

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

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

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

Finally, MM follows the literature about mixin modules [16, 6, 45] in its set of operators, including composition $e_1 \circ e_2$, closure close $e$, freezing $e \mapsto X$, projection $e_{[X_1 \ldots X_n]}$, deletion $e_{\mid X_1 \ldots X_n}$, showing $e_{\cdot X_1 \ldots X_n}$, hiding $e_{\cdot x_1 \ldots x_n}$, and renaming $e_{[X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n]}$. There is a new operator called splitting $e_{X \cdot Y}$. We let $op$ range over the set of operators (see figure 3.2), and denote by $op[e]$ the application of $op$ to the expression $e$.

**Syntactic correctness**  Renamings $r = (X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n)$, inputs $t = (X_1 \mapsto x_1 \ldots X_n \mapsto x_n)$, records $s = (X_1 = e_1 \ldots X_n = e_n)$, bindings $b = (x_1 = e_1 \ldots x_n = e_n)$, are required to be finite maps: a renaming is a finite map from names to names, an input is a finite map from names to variables, a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice. Renamings and inputs are required to be injective. Outputs $o = (d_1 \ldots d_m)$ are required not to define the same name twice, and not to define the same variable twice. Structures are required not to define the same name twice and not to define the same variable twice. Fake dependencies in a definition must be bound in the same structure.
$$x \in \text{Vars}$$  \hspace{1cm} \text{Variable} \\
$$X \in \text{Names}$$  \hspace{1cm} \text{Name} \\

**Expression:**  
\[ e ::= x \]
\[ \{ X_1 = e_1 \ldots X_n = e_n \} \]
\[ e.X \]
\[ \text{let rec } x_1 = e_1 \ldots x_n = e_n \text{ in } e \]
\[ \langle X_1 \triangleright x_1 \ldots X_n \triangleright x_n; d_1 \ldots d_m \rangle \]
\[ e_1 + e_2 \mid \text{close } e \mid e ! X \]
\[ e_{|X_1,\ldots,X_n} \mid e_{|X_1,\ldots,X_n} \]
\[ e_{[X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n]} \]
\[ e_{X \triangleright Y} \]

**Definition:**  
\[ d ::= X[x_1 \ldots x_n] \triangleright x = e \]
\[ [x_1 \ldots x_n] \triangleright x = e \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s ::= X_1 = e_1 \ldots X_n = e_n$</td>
<td>Record</td>
</tr>
<tr>
<td>$b ::= x_1 = e_1 \ldots x_n = e_n$</td>
<td>Binding</td>
</tr>
<tr>
<td>$t ::= X_1 \triangleright x_1 \ldots X_n \triangleright x_n$</td>
<td>Input (injective)</td>
</tr>
<tr>
<td>$o ::= d_1 \ldots d_2$</td>
<td>Output</td>
</tr>
<tr>
<td>$r ::= X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n$</td>
<td>Renaming (injective)</td>
</tr>
<tr>
<td>$\phi[p][e] ::= e.X$</td>
<td>Record selection</td>
</tr>
<tr>
<td>$\mid \text{close } e \mid e ! X$</td>
<td>Closure, freezing</td>
</tr>
<tr>
<td>$e_{</td>
<td>X_1,\ldots,X_n} \mid e_{</td>
</tr>
<tr>
<td>$e_{X_1,\ldots,X_n} \mid e_{X_1,\ldots,X_n}$</td>
<td>Showing, hiding</td>
</tr>
<tr>
<td>$e_{[X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n]}$</td>
<td>Renaming</td>
</tr>
<tr>
<td>$e_{X \triangleright Y}$</td>
<td>Splitting</td>
</tr>
</tbody>
</table>

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

**Figure 3.2:** Meta-variables and notations

---

**Figure 3.1:** Syntax of MM
Value: \[ v ::= x \mid \{ s_v \} \]
\[ \mid \langle X_1 \triangleright x_1 \ldots X_n \triangleright x_n ; d_1 \ldots d_n \rangle \]

Answer: \[ a ::= v \mid \text{let rec } b_v \text{ in } v \]

Value sequence: \[ s_v ::= X_1 = v_1 \ldots X_1 = v_1 \]
\[ b_v ::= x_1 = v_1 \ldots x_n = v_n \]

**Figure 3.3: Values in MM**

\[ \langle \tau_1 ; o_1 \rangle \preceq \langle \tau_2 ; o_2 \rangle \text{ means } \begin{cases} \langle \tau_1 ; o_1 \rangle = \langle \tau_2 ; o_2 \rangle & \text{and} \\ \langle \tau_2 ; o_2 \rangle = \langle \tau_1 ; o_1 \rangle & \text{.} \end{cases} \]
\[ \langle \tau_1 ; o_1 \rangle = \langle \tau_2 ; o_2 \rangle \text{ means that for all } (L \triangleright x) \in \text{dom}(\langle \tau_1 ; o_1 \rangle), \]
\[ x \in FV(o_2) \cup \text{Variables}(\langle \tau_2 ; o_2 \rangle) \Rightarrow (L \triangleright x) \in \text{dom}(\langle \tau_2 ; o_2 \rangle) \text{ and } L \in \text{Names}. \]

**Figure 3.4: Definition of \(\preceq\)**

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

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

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

### 3.2 Semantics

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

**Values** As defined in figure 3.3, an MM value is either a variable \(x\), or an evaluated record \(\{ X_1 = v_1 \ldots X_1 = v_1 \}\), or a structure \(\langle \tau ; o \rangle\). A valid result of the evaluation of an MM expression is a value, possibly surrounded by an evaluated binding. It thus has the shape let rec \(x_1 = v_1 \ldots x_n = v_n\) in \(v\). The meta-variables \(s_v\) and \(b_v\) respectively range over evaluated record sequences and bindings.
Figure 3.5: Computational contraction relation
The contraction relation  The contraction relation is defined by the rules in figure 3.5, where for any sets $P_1 \ldots P_n, P_1 \perp \ldots \perp P_n$ means that the $P_i$'s are pairwise disjoint.

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

$$L ::= \{S\} | op[p] | \square + e | v + \square$$

$S ::= \sigma_v, X = \square, s.$

Rule Lift states that an expression of the shape $L \llbracket \text{let rec } b \text{ in } e \rrbracket$ evaluates to let rec $b$ in $L[e]$, provided no variable capture occurs.

The record selection rule Select straightforwardly describes the selection of a record field.

The rules for mixin deletion Delete and projection Project are dual. Rule Delete describes how MM deletes a finite set of names $X_1 \ldots X_n$ from a structure $\langle \iota; o \rangle$. First, $o$ is restricted to the other definitions, to obtain $o\{\langle X_1 \ldots X_n \rangle\}$ (which is a shorthand for $o\{\langle X_1 \ldots X_n \rangle \times \text{vars}\}$). Second, the removed definitions remain bound as inputs, by adding the corresponding inputs to $\iota$.

Rule Project describes how MM projects a mixin to some finite set of names $X_1 \ldots X_n$ from a structure $\langle \iota; o \rangle$. First, $o$ is restricted to the corresponding definitions and to the local ones, to obtain $o\{\langle \langle X_1 \ldots X_n \rangle \rangle\}$ (which is a shorthand for $o\{\langle X_1 \ldots X_n \rangle \times \text{vars}\}$). Then, the removed definitions remain bound as inputs, by adding the corresponding inputs to $\iota$.

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

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

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

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

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

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

The Sum rule defines the composition of two structures $\langle \iota_1; o_1 \rangle$ and $\langle \iota_2; o_2 \rangle$. The result is a structure $\langle \iota; o \rangle$, defined as follows. $\iota$ is the union of $\iota_1$ and $\iota_2$, where names defined in $o_1$ or $o_2$ are removed. $o$ is defined as the concatenation of $o_1$ and $o_2$. The side condition $\langle \iota_1; o_1 \rangle \odot \langle \iota_2; o_2 \rangle$ checks that both structures agree on bound variables, and that no free variable is captured. It is defined in figure
Evaluation context:
\[
E ::= F \mid \text{let rec } b_v \text{ in } F \mid \text{let rec } B \{ F \} \text{ in } e
\]

Lift context:
\[
L ::= \{ S \} \mid op[\square] \mid \square + e \mid v + \square
\]

Multiple lift context:
\[
F ::= \square \mid L \{ F \}
\]

Binding context:
\[
B ::= b_v, x = \square, b
\]

Record context:
\[
S ::= s_v, X = \square, s
\]

Figure 3.6: Evaluation contexts

\[
\text{(let rec } b_v \text{ in } F)(x) = b_v(x) \quad \text{(EA)} \quad \text{(let rec } b_v, y = F, b \text{ in } e)(x) = b_v(x) \quad \text{(IA)}
\]

Figure 3.7: Access in evaluation contexts

3.4, where \(\text{dom}(\zeta; o) = \zeta \cup \text{dom}(o)\), and \(\text{Variables}(\zeta; o)\) denotes \(\text{cod}(\zeta) \cup \{ x \mid (L, x) \in \text{dom}(o) \}\). Lastly, \(o_1\) and \(o_2\) are required not to define the same names.

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

The reduction relation The reduction relation is defined by the rules in figure 3.8, using notions defined in figures 3.6 and 3.7.

Rule \text{Context} extends the contraction relation to any evaluation context. Evaluation contexts are defined in figure 3.6. We call a multiple lift context \(F\) a series of nested lift contexts. An evaluation context \(E\) is a multiple lift context, possibly inside a partially evaluated binding, or under a fully evaluated binding. This unusual formulation of evaluation contexts is intended to enforce determinism of the reduction relation. The idea is that evaluation never takes place inside or under a let rec, except the topmost one. Other bindings inside the expression first have to be lifted to the top by rule \text{Lift}, and then merged with the topmost let rec if any, by rules \text{EM} and \text{IM}. In the case where the topmost binding is of the shape \(b_v, x = (\text{let rec } b_1 \text{ in } e), b_2\), rule \text{IM} allows to merge \(b_1\) with the current binding. When an inner binding has been lifted to the top, if
there is already a topmost binding, then the two bindings are merged together by rule EM. As a result, when the evaluation encounters a binding, it is always possible to lift it up to the top and then merge it with the topmost binding if any.

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

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

In MM the value of a variable is copied only when needed for the application of an operator, or for composition. The value of a variable $x$ is found in the current evaluation context, by looking for the first binding of $x$ above the calling site, as formalized by the notion of access in evaluation contexts in figure 3.7. There are two kinds of accesses.

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

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

In figure 3.7, the capture conditions are formalized with the Capt function. $\text{Capt}_{\square}(E)$ is the set of bound variables above $\square$ in $E$. If $\square$ is filled with another variable, then it is free in the obtained expression.

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

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

More formally, the dependency graph of an output is defined in figure 3.9. For each pair of definitions $L[y^*] \triangleright x = e$ and $L[z^*] \triangleright x' = e'$ in $o$, there may be two kinds of edges.

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

$$\begin{align*}
\text{Degree}(x, (\emptyset; o)) &= \emptyset \\
\text{Degree}(x, (s_v)) &= \emptyset \\
\text{Degree}(x, e) &= \emptyset \text{ otherwise.}
\end{align*}$$

The $\text{Degree}$ function is simple, and could be extended as in [12, 45].
\[(L[y^*] \triangleright x = e) \in o \quad (L'[z^*] \triangleright x' = e') \in o \quad \chi = \text{Degree}(x', e)\]

\[x' \xrightarrow{\chi, o} x\]

\[(L[x_1 \ldots x_n] \triangleright x = e) \in o \quad (L'[z^*] \triangleright x_i = e') \in o\]

\[x_i \xrightarrow{\Theta, o} x\]

**Figure 3.9: Dependencies in an output**

\[
\frac{x \xrightarrow{\chi, o} z \quad z \xrightarrow{\chi, o} y}{x \xrightarrow{\chi, o} y}
\]

\[
\frac{x \xrightarrow{\chi, o} y}{x \xrightarrow{\chi, o} y}
\]

**Figure 3.10: Transitive closure of \(\rightarrow\)**

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

The transitive closure of this relation is defined in figure 3.10, by defining the degree of a path as the degree of its last edge. The relation \(\xrightarrow{\Theta, o}^+\) gives a conservative approximation of which definition needs the value of which other one in \(\text{Bind}(\overline{\pi})\). Reordering \(o\) according to \(\rightarrow_o\) it is not enough though, because the generated binding might be syntactically incorrect. Indeed, it is forbidden to make forward references to definitions of unpredictable shape inside a binding. Strict forward references to definitions of unpredictable shape already correspond to edges labeled \(\Theta\) in \(\rightarrow_o\), and are therefore taken into account when reordering according to \(\xrightarrow{\Theta, o}^+\). Weak forward references to definitions of unpredictable shape correspond to edges labeled \(\Theta\) in \(\rightarrow_o\), and are therefore not taken into account when reordering according to \(\xrightarrow{\Theta, o}^+\). Let \(\succ_{o} = \{ (x_1, x_2) \mid x_1 \xrightarrow{\Theta, o} x_2, o(x_1) \notin \text{Predictable} \}\). This relation exactly puts weak references to definitions of unpredictable shape in the right order.

We define the binary relation \(\succ_o\) by the lexical ordering \(\succ_o = ((\xrightarrow{\Theta, o}^+ \cup \succ_o)^+)\), where \(\succ_o\) is the initial ordering in \(o\). If \(\succ_o\) contains no cycle, \(o\) is said correct. This is written \(\vdash o\). In this case, \(\overline{\sigma}\) denotes \(o\) reordered by \(\succ_o\).
Chapter 4

Static semantics

4.1 Type system

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

Types are defined in figure 4.1. There are only two kinds of types, record types \( \{O\} \) and mixin types \( \langle I; O; G \rangle \), where \( I \) and \( O \) range over finite maps from names to types and \( G \) is a finite graph over names, labeled by degrees. Such a graph is called an abstract dependency graph. (Remember that dependency graphs over the whole set of nodes are called concrete.) An environment \( \Gamma_0 \) is a finite map from variables to types. We write \( \Gamma(\Gamma') \) for the map where the bindings of \( \Gamma' \) have overridden the ones from \( \Gamma \).

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

\[
N_1 \xrightarrow{\chi_1} N_2 \quad \rightarrow \quad N_1 \xrightarrow{\chi_1 \land \chi_2} N_2
\]

where \( \land \) gives the most dangerous of two degrees:

\[
\chi_1 \land \chi_2 = \ominus \text{ if } \chi_1 = \chi_2 = \ominus \\
\chi_1 \land \chi_2 = \ominus \text{ otherwise}
\]

In figure 4.2, the type system is defined by means of a set of inference rules.

The first rule T-STRUCT concerns the typing of basic structures \( \langle i; o \rangle \). Given an input \( i \) (which is arbitrary here, we do not consider type inference or type-checking issues) corresponding to \( i \), and a type environment \( \Gamma_0 \) corresponding to \( o \), it checks that the definitions in \( o \) indeed have the types mentioned in \( \Gamma_0 \).

<table>
<thead>
<tr>
<th>( M \in \text{Types} )</th>
<th>::=</th>
<th>( {O} \mid \langle I; O; G \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I, O )</td>
<td>( \in )</td>
<td>Names ( \stackrel{\text{Fin}}{\rightarrow} ) Types</td>
</tr>
<tr>
<td>( G )</td>
<td>( \subseteq \text{Fin} )</td>
<td>( {X \xrightarrow{\chi} Y \mid X, Y \in \text{Names}, \chi \in \text{Degrees}} )</td>
</tr>
<tr>
<td>( \Gamma )</td>
<td>( \in )</td>
<td>Vars ( \stackrel{\text{Fin}}{\rightarrow} ) Types</td>
</tr>
</tbody>
</table>

Figure 4.1: Types
Expressions:

\[
\text{dom}(\ell) = \text{dom}(I) \quad \vdash I \quad \vdash \Gamma_e \quad \vdash \rightarrow_{(\ell; o)} \quad \Gamma(I \circ \ell \downarrow \Gamma_e) \vdash o : \Gamma_e \quad \text{(T-STRUCT)}
\]

\[
\frac{I_1 \cup O_1 \cup I_2 \cup O_2 \vdash G_1 \cup G_2 \quad \Gamma \vdash e_1 : \langle I_1; O_1; G_1 \rangle \quad \Gamma \vdash e_2 : \langle I_2; O_2; G_2 \rangle}{\Gamma \vdash e_1 + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \cup O_2; G_1 \cup G_2 \rangle} \quad \text{(T-SUM)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad X \in \text{dom}(O)}{\Gamma \vdash e!X : \langle I; O; [G ! X] \rangle} \quad \text{(T-FREEZE)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle}{\Gamma \vdash e : \langle I; O; G \rangle \quad \Gamma \vdash \text{close} e : \{O\}} \quad \text{(T-CLOSE)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad X_1 \ldots X_n \in \text{dom}(O)}{\Gamma \vdash e_{X_1 \ldots X_n} : \langle I \cup O; \{X_1 \ldots X_n\}; G; \{X_1 \ldots X_n\} \rangle} \quad \text{(T-PROJECT)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad X_1 \ldots X_n \in \text{dom}(O)}{\Gamma \vdash e_{X_1 \ldots X_n} : \langle I \cup O; \{X_1 \ldots X_n\}; G \setminus \{X_1 \ldots X_n\} \rangle} \quad \text{(T-DELETE)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad \{X_1 \ldots X_n\} \in \text{dom}(O)}{\Gamma \vdash e : \langle I; O; G \rangle \quad \{X_1 \ldots X_n\} \in \text{dom}(O)} \quad \text{(T-HIDE)}
\]

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad \{X_1 \ldots X_n\} \in \text{dom}(O)}{\Gamma \vdash e : \langle I; O; G \rangle \quad \{X_1 \ldots X_n\} \in \text{dom}(O)} \quad \text{(T-SHOW)}
\]

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

\[
\frac{\Gamma \vdash e : \langle I; O; G \rangle \quad Y \notin \text{dom}(I) \cup \text{dom}(O)}{\Gamma \vdash e_{X \rightarrow Y} : \langle I \cup \{X : O(X)\}; O\{X \rightarrow Y\}; G_{X \rightarrow Y} \rangle} \quad \text{(T-SPLIT)}
\]

\[
\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash e_i : M_i}{\Gamma \vdash \{X_1 = e_1 \ldots X_n = e_n\} : \{X_1 : M_1 \ldots X_n : M_n\}} \quad \text{(T-RECORD)}
\]

\[
\frac{\vdash b \quad \vdash b \quad \Gamma(b) \vdash b : \Gamma_b \quad \Gamma(b) \vdash e : M}{\Gamma \vdash \text{let rec} \ b \ \text{in} \ e : M} \quad \text{(T-LETREC)}
\]

\[
\frac{\Gamma \vdash e : \emptyset \quad \Gamma \vdash \ell [x^*] : x = e, o : \{x : M\} \cup \Gamma_e}{\Gamma \vdash (L[x^*] \triangleright x = e, o : \{x : M\} \cup \Gamma_e} \quad \frac{\Gamma \vdash e : M \quad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x = e, b) : \{x : M\} \cup \Gamma_b} \quad \text{(T-VARIABLE)}
\]

Sequences:

\[
\frac{\Gamma \vdash e : M \quad \Gamma \vdash o : \Gamma_e}{\Gamma \vdash (L[x^*] \triangleright x = e, o : \{x : M\} \cup \Gamma_e} \quad \frac{\Gamma \vdash e : M \quad \Gamma \vdash b : \Gamma_b}{\Gamma \vdash (x = e, b) : \{x : M\} \cup \Gamma_b}
\]

Figure 4.2: Type system
**Figure 4.3: Graph operations**

\[
\begin{align*}
\chi &= \text{Degree}(x', e) & (L', x') \in \text{dom}(\langle t; o \rangle) & (L[x^*] \triangleright x = e) \in o \\
\text{Node}(L', x') & \rightarrow_{\langle t; o \rangle} \text{Node}(L, x) \\
(L_i, x_i) \in \text{dom}(\langle t; o \rangle) & (L[x_1 \ldots x_n] \triangleright x = e) \in o \\
\text{Node}(L_i, x_i) & \rightarrow_{\langle t; o \rangle} \text{Node}(L, x)
\end{align*}
\]

**Figure 4.4: Dependencies in a structure**
The condition \( \vdash \rightarrow_{(i,o)} \) requires some explanation. We saw in section 3.2 that dependencies in an output are represented by its dependency graph \( \rightarrow_c \). For structures (which are incomplete outputs), the corresponding notion is the concrete dependency graph. A concrete dependency graph is a graph over nodes. A node \( N \) is an element of \( \text{Nodes} = \text{Vars} \cup \text{Names} \). The dependency graph of a structure is defined in figure 4.4. It records dependencies in the structure (as was done for outputs), but takes external names into account, when possible. Named definitions are represented by a name, and local definitions are represented by their variables. In order for types not to mention local components, we introduce a \emph{lift} operation \( [\vdash_{i,o}] \), which, as described in figure 4.3, first ensures to keep track of local components by shifting their dependencies to the next exported components, and then erases them. The result is an abstract dependency graph.

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

**Definition 4 (Correct graphs)** A graph \( \rightarrow \) is correct iff \( \bigcirc \vdash^+ \) is an ordering on its nodes (written \( \vdash \rightarrow \)).

**Definition 5 (Well-formed types)** Figure 4.5 defines the sets of well-formed types on inputs (or outputs), as the least relation respecting the rules. A mixin type \( (I;O;G) \) must import and define disjoint sets of names, the targets of \( G \) must be defined, and \( G \) must be correct.

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

The third rule T-FREEZE introduces a new operation \( G!X \triangleright x \) on abstract graphs, which is again defined in figure 4.3. To freeze a name \( X \), it first replaces \( X \) with a fresh local variable \( x \), making the graph temporarily non-abstract. Then, it adds a strict link from \( x \) to \( X \). This follows closely the semantics of freezing from figure 3.5, first making all other components call the local component \( x \) instead of \( X \), and then re-exporting \( X \) as \( x \) exactly. The link is forced to be a strict one by hypothesis 2.

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

The mixin projection rule T-PROJECT, exactly as the corresponding contraction rule, keeps in the output types only the selected ones, reporting the other ones in the input types. The abstract graph is modified accordingly by the operation \( G_{(X_1 \ldots X_n)} \), which removes the edges leading to unselected components. The T-DELETE rule is its dual again.

The T-HIDE removes the given names from the output. Additionally, it acts on the abstract graph \( G \) as described in figure 4.3. It first replaces the given names by fresh variables, and then lifts the result, in order to obtain an abstract graph. Rule T-SHOW is its dual, as expected.
Rule T-RNAME, given a mixin \( e \) of type \( \langle I; O; G \rangle \), deduces that \( e \) renamed by \( r \) has the same type, with input \( I \) and output \( O \) redirected to use the new names \((\text{cod}(r))\). As the contraction rule RENAME, it makes use of the \( r_N \) function, composed with \( I \) and \( O \). The abstract graph is renamed as well.

Given an expression \( e \) of type \( \langle I; O; G \rangle \), according to rule T-SPLIT, the type of \( e_{X\to Y} \) is as follows. \( X \) is added to the input, with the type it had in \( O \). \( X \) is renamed to \( Y \) in the output. The graph \( G \) is modified according to figure 4.3. \( G_{\{X\to Y\}} \) is the set of edges leading to \( X \) in \( G \). Basically, these edges are redirected to \( Y \).

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

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

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

### 4.2 A theory of dependency graphs and degrees

For proving the soundness of MM, we will have to prove some properties of the operations we use on dependency graphs. Such operations will be used later in this thesis, so we abstract over the current definitions in order to make the proofs valid for further use.

We begin with a characterization of the properties needed for degrees.

**Definition 6 (Degrees)** A set Degrees has a structure of degrees iff it is a complete lattice, and its elements are divided into positive and negative elements, compatible with the ordering.

We fix an arbitrary structure of degrees Degrees for this section, whose elements are denoted by \( \chi \), ordering is denoted by \( \succeq \), greatest lower bound operation is denoted by \( \land \). We denote by Positive and Negative the sets of positive and negative degrees, respectively. The compatibility condition means that for all \( \chi_1 \in \text{Positive} \) and \( \chi_2 \in \text{Negative} \), we have \( \chi_1 \succeq \chi_2 \).

**Definition 7 (Dependency graph)** A dependency graph is a finite, oriented graph, labeled with degrees.

The nodes of dependency graphs are not relevant to the properties we want to establish, so we do not constrain them at all. We denote them by \( N \), and denote finite sets of them by \( \mathcal{N} \). We denote the set of nodes of a graph \( \rightarrow \) by Nodes(\( gD \)).

**Definition 8 (Transitive closure)** We define the transitive closure on dependency graphs as the fixed-point of the operation that adds an edge \( N_1 \xrightarrow{\Delta} N_3 \) for each pair of edges \( N_1 \xrightarrow{\Delta} N_2 \) and \( N_2 \xrightarrow{\Delta} N_3 \) in its argument \( \rightarrow \).

This fixed-point is always well-defined, since the considered operation does not introduce any degree, so the number of edges of the generated graphs is bounded. The transitive closure of a graph \( \rightarrow \) is written \( \rightarrow^* \). Its reflexive transitive closure is written \( \rightarrow^+ \).

Some notions on *paths* are defined as follows.
Definition 9 (Paths) A path of the dependency graph $\rightarrow$ is a possibly empty list of consecutive edges. Its length is its number of edges. If its length is strictly positive, then its degree is defined as the degree of its last edge. A cycle is a non-empty path whose the source and target nodes are the same.

We denote paths by $\delta$. The concatenation of two consecutive paths is written $\delta_1; \delta_2$. For a dependency graph $\rightarrow$, a path is also an edge of $\rightarrow^*$. The degree of a non-empty path is defined as the degree of its last edge. We write $N_1 \xrightarrow{\delta_1^+} N_2$ for a non-empty path of degree $\chi$. Also, the concatenation of a non-empty path $N_1 \xrightarrow{\delta_1^+} N_2$ and a possibly empty path $\delta$ from $N_2$ to $N_3$ is written $N_1 \xrightarrow{\delta_1^+ \delta} N_3$, where $\chi_2 = \chi_1$ if $\delta$ is empty, and the degree of $\delta$ otherwise. Finally, when the two ends of successive paths or edges are syntactically the same, we merge them. For instance, the concatenation above could have been written $N_1 \xrightarrow{\delta_1^+} N_2 \xrightarrow{\delta_2^*}$.

Let us introduce two notions of correctness for dependency graphs. It relies on the notion of a safe cycle: a cycle is safe if all its edges are labeled with positive degrees. Otherwise, the cycle is said unsafe.

Definition 10 (Correctness) A dependency graph $\rightarrow$ is said correct if its transitive closure does not contain any unsafe cycle. We write it $\vdash \rightarrow$.

This notion is related to the following notion of ordered correctness, which relies on an order on nodes. Orders on nodes are denoted by the symbol $\triangleright$. Their strict versions are denoted by $\triangleright^>$. For any dependency graph $\rightarrow$, let $\text{Negative}_\rightarrow$ be the set of edges of $\rightarrow$ that are labeled with negative edges.

Definition 11 (Ordered correctness) A dependency graph $\rightarrow$ is correct with respect to the order $\triangleright$, or respects the order $\triangleright$, if $\rightarrow^+$ is compatible with $\triangleright$. We write it $\vdash (\rightarrow, \triangleright)$ (or $\vdash (\rightarrow, \triangleright^>)$).

We have the following equivalence.

Property 1 $\vdash \rightarrow$ iff there exists an ordering $\triangleright$ on Nodes($\rightarrow$) such that $\vdash (\rightarrow, \triangleright)$.

For proving it, we introduce the notion of a backward edge and a backward path.

Definition 12 (Backward edges and paths) Given a dependency graph $\rightarrow$ and an order $\triangleright$ on nodes, an edge $N_1 \xrightarrow{\delta} N_2$, or a path $N_1 \xrightarrow{\delta^*}$ is said backward if $N_2 \triangleright N_1$.

Proof

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

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

Now, we prove two properties that we will directly use later in soundness proofs. Their names are related to the reduction rules they correspond to. Each of them is associated with a picture that is supposed to help the reader understand them.

Property 2 (External merge)
4.3 Graph soundness

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

Hypothesis 1 (Shape)

- \(x \notin \text{Predictable}\).
- \(\langle t; o \rangle \in \text{Predictable} \text{ and } \{s_v\} \in \text{Predictable}\).
- Let \(\sigma\) be a variable renaming. \(e[\sigma] \in \text{Predictable} \text{ iff } e \in \text{Predictable}\).
- If \(E[x] \in \text{Predictable}, \text{ then } E[v] \in \text{Predictable}, \text{ for all } v\).
- If \(e \rightarrow e'\) and \(e \in \text{Predictable}, \text{ then } e' \in \text{Predictable}\).
- If \(e \in \text{Predictable} \text{ and } e' \in \text{Predictable}, \text{ then for any context } E, \ E[v] \in \text{Predictable} \text{ iff } E[e'] \in \text{Predictable} \).

We require the degree function to meet the following condition.

Hypothesis 2 (Degree function)

- If \(\text{Degree}(x, e) = \emptyset\), then \(e \in \text{Predictable}\).
- If \(e \rightarrow e'\) and \(\text{Degree}(x, e) \neq \emptyset\), then \(\text{Degree}(x, e') \neq \emptyset\).
- If \(x \in \text{FV}(e) \setminus \text{Cap}_t(E[N]), \text{ then } \text{Degree}(x, E[N[e]]) = \emptyset\).
- If \(y \notin \text{FV}(v) \setminus \text{Cap}_t(E[F]), \text{ then } \text{Degree}(y, E[v]) = \text{Degree}(y, F)\).
- If for all \(x \in \text{FV}(e), \text{Degree}(x, e') \leq \text{Degree}(x, e), \text{ then for any context } E, \text{ for any } x \in \text{FV}(E[e]), \text{Degree}(x, E[e']) \leq \text{Degree}(x, E[e])\).

\[\forall x \notin \text{dom}(b). X \neq Y, \forall \chi \in \{\chi_{\text{H}(X \geq_{(X \neq x)} N, o = (Output(b), Y \triangleright \ast = e)}\}, \text{Degree}(x, \text{let rec } b \text{ in } e) \leq \chi.\]

4.3.1 Modeling the reduction with graphs

Definition 13 (Mixin redex) Mixin redexes \(e \uparrow\) are defined by

\[e \uparrow := \langle t_1; o_1 \rangle + \langle t_2; o_2 \rangle | op[\langle t; o \rangle].\]

The graph operations on abstract graphs defined in figure 4.3 are trivially generalized to concrete graphs. These operations are used to guess the concrete graph of a mixin redex.

Definition 14 (Graph of a mixin redex)

\[\rightarrow \langle t_1; o_1 \rangle + \langle t_2; o_2 \rangle = \rightarrow \langle t_1; o_1 \rangle + \rightarrow \langle t_2; o_2 \rangle\]

\[\rightarrow_{op}[\langle t; o \rangle] = op(\rightarrow \langle t; o \rangle)\]

Proposition 1 (Graphs operations model contraction) If \(e \uparrow \sim e\), then \(\rightarrow e_1 = \rightarrow e\).

Proof By case analysis on the reduction.
Sum. We have \( e'_1 = \langle \ell_1; o_1 \rangle + \langle \ell_2; o_2 \rangle \) and \( e = \langle \ell_1; o_1, o_2 \rangle \), with \( \ell = \{ \ell_1 \cup \ell_2 \} \setminus \text{Input}(o_1, o_2) \). Trivially, \( \to_{e'_1} = \to_{\langle \ell_1; o_1 \rangle} \cup \to_{\langle \ell_2; o_2 \rangle} = \to_{\langle \ell_1; o_1 \rangle} \cup \to_{\langle \ell_2; o_2 \rangle} \) by \( \langle \ell_1; o_1 \rangle \circ \langle \ell_2; o_2 \rangle \). Then, \( \to_{\ell_1} \cup \to_{\ell_2} = \to_{\langle \ell_1; o_1 \rangle} \cup \to_{\langle \ell_2; o_2 \rangle} \).

Freeze. Let \( e'_1 = \langle \ell_1; o_1, X[y^*] \triangleright x = f, o_2 \rangle \). X, and \( e = \langle \ell_1; o_1, X[y^*] \triangleright x = f, o_2 \rangle \). Its graph is exactly the same as the one of \( e \) except that instead of the node \( X \), we find the node \( \text{Node}(\triangleright x) \), which is \( x \). Then, append the component \( X \triangleright y = x \) with a fresh \( y \). This adds a strict dependency from \( X \) to \( x \), so the result is exactly \( \to_{e'_1} \).

Other cases similar.

\( \square \)

4.3.2 Subject contraction for graphs

The goal of this section is to ensure that abstract graphs detect all errors in the underlying concrete graphs. We write \( \delta \) for paths in graphs. The minimum degree of a path \( \delta = X \xrightarrow{\lambda_1} N_1 \ldots N_{n-1} \xrightarrow{\lambda_n} Y \) is 
\[
\chi = \bigwedge_{1 \leq i \leq n} \chi_i.
\]

Proposition 2 (Lift preserve paths between names) Let \( \delta \) be a path for the \( \to \) relation, starting with name \( X \), ending with name \( Y \), and having minimum degree \( \chi \). Let \( G = [\to] \). There exists a path from \( X \) to \( Y \) in \( G \), with the same minimum degree.

Proof Let \( \delta = N_0 \xrightarrow{\lambda_1} N_1 \ldots N_{n-1} \xrightarrow{\lambda_n} N_n \). We proceed by induction on the number of names in the path.

Base. Two names, \( \delta = X \xrightarrow{\lambda_1} x_1 \ldots x_n \xrightarrow{\lambda_n} Y \). An easy induction on \( n \) proves that \( X \xrightarrow{\lambda_n} Y \), and therefore \( (X,Y,\chi) \in G \).

Induction. By induction hypothesis.

\( \square \)

Corollary 1 If \( \xrightarrow{\Theta^+} \) has a cycle with at least one name, then \( [\to]^{\Theta^+} \) also has one.

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

Proposition 3 (Lift commutes with operators) Let \( \to_1, \to_2 \), and \( \to \) be concrete dependency graphs (i.e. graphs over Nodes).

- If the variables from \( \to_1 \) and the ones from \( \to_2 \) are disjoint, then \( [\to_1 \cup \to_2] = [\to_1 \cup [\to_2] \).
- \( [ap[\to]] = ap[[\to]] \), for \( ap \in \{ !X, X, X \triangleright Y \} \) (with \( \text{cod}(r) \perp \text{Nodes}(\to) \)).

Proof

Sum. It is obvious that \( [\to_1] \cup [\to_2] \subset [\to_1 \cup \to_2] \), since \( \to_1 \cup \to_2 \) and lift is monotone.

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

70
**Freeze.** Let \( x \) be a fresh variable. By definition, we have to prove that \( \llbracket \rightarrow !X \triangleright x \rrbracket = \llbracket \llbracket \rightarrow !X \triangleright x \rrbracket \).

Let 
\[
\rightarrow_1 = \rightarrow !X \triangleright x \quad \text{and} \quad \rightarrow_2 = \llbracket \rightarrow !X \triangleright x \rrbracket
\]

\[
\rightarrow'_1 = \llbracket \rightarrow_1 \rrbracket 
\quad \rightarrow'_2 = \rightarrow_2 !X \triangleright x 
\quad \rightarrow''_2 = \llbracket \rightarrow'_2 \rrbracket
\]

First, notice that both in \( \rightarrow'_1 \) and \( \rightarrow'_2 \), no edge starts from \( X \), and edges arriving to \( X \) come from paths to \( X \) through \( x \) with degree \( \odot \) in \( \rightarrow_1 \) and \( \rightarrow'_2 \), respectively, so they have degree \( \odot \).

- \( \rightarrow''_2 \subset \rightarrow'_1 \).

  - Let \( Y \xrightarrow{\xi} X \), with \( X \neq Y \). Necessarily, \( \xi = \odot \).
    
    This implies that there exists a path of \( \rightarrow'_2 \) of the shape
    
    \[
    Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_2 X,
    \]
    
    because \( x \) is the only variable in \( \rightarrow'_2 \).

    \( n \) could be zero, in which case we would have \( Y \overset{\odot}{\longrightarrow}_2 X \).

    But this means that we have \( Y \xrightarrow{\xi_0} X \xrightarrow{\xi_1} X \ldots \xrightarrow{\xi_n} X \).

    So by definition of \( \llbracket \rrbracket \), we have \( Y \xrightarrow{\xi_0} X \xrightarrow{\xi_1} X \ldots \xrightarrow{\xi_n} X \).

    So, we have \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 X \),

    and therefore \( Y \overset{\odot}{\longrightarrow}_1 X \).

  - Let \( Y \xrightarrow{\xi} Z \), with \( Y \) and \( Z \) different from \( X \). Then
    
    \[
    Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} z Z,
    \]
    
    because \( x \) is the only variable in \( \rightarrow'_2 \).

    We have \( \chi = \bigwedge_{0 \leq i \leq n} \chi_i \) \( n \) could possibly be 0, in which case the path would rather look like \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 Z \).

    We can deduce \( Y \xrightarrow{\xi_0} X \xrightarrow{\xi_1} X \ldots X \xrightarrow{\xi_n} Z \),

    so \( Y \xrightarrow{\xi_0} X \xrightarrow{\xi_1} X \ldots X \xrightarrow{\xi_n} Z \),

    and therefore \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 Z \).

    So we have \( Y \xrightarrow{\xi} Z \).

- \( \rightarrow'_1 \subset \rightarrow''_2 \).

  - Let \( Y \xrightarrow{\xi} X \), with \( Y \neq X \). We have
    
    \[
    Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 X,
    \]
    
    where for all \( i \), \( \xrightarrow{\xi_i} \) does not go through \( x \).

    As above, we have \( \chi = \odot \) and \( n \) could possibly be 0, in which case the path would rather look like \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 X \).

    This implies that \( Y \xrightarrow{\xi_0} X \xrightarrow{\xi_1} X \ldots X \xrightarrow{\xi_n} X \).

    Therefore, \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 X \),

    and so \( Y \overset{\odot}{\longrightarrow}_2 X \).

  - Let \( Y \xrightarrow{\xi} Z \), with \( Y \) and \( Z \) different from \( X \).

    We deduce \( Y \xrightarrow{\xi_0} x \xrightarrow{\xi_1} x \ldots x \xrightarrow{\xi_n} x \overset{\odot}{\longrightarrow}_1 Z \),

    where for all \( i \), \( \xrightarrow{\xi_i} \) does not go through \( x \).

    As above, we have \( \chi = \bigwedge_{0 \leq i \leq n} \chi_i \) and \( n \) could possibly be 0, in which case the path would rather look like \( Y \xrightarrow{\xi_0} Z \).
Then, \( Y \xrightarrow{X, \rho} X \xrightarrow{X, \rho} X \ldots X \xrightarrow{X, \rho} Z \),
and so \( Y \xrightarrow{X, \rho} X \xrightarrow{X, \rho} X \ldots X \xrightarrow{X, \rho} Z \),
which leads to \( Y \xrightarrow{X, \rho} x \xrightarrow{X, \rho} x \ldots x \xrightarrow{X, \rho} Z \),
and so \( Y \xrightarrow{X, \rho} Z \).

**Other cases.** Easy.

\[ \square \]

**Corollary 2** If \( \vdash \rightarrow \), then \( \vdash op(\rightarrow) \). If \( \vdash \rightarrow_1, \vdash \rightarrow_2 \), Variables(\( \rightarrow_1 \)) \(\perp\) Variables(\( \rightarrow_2 \)), and \( \vdash [\rightarrow_1 \cup [\rightarrow_2] \), then \( \vdash \rightarrow_1 \cup \rightarrow_2 \).

**Proof**

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

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

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

\[ \square \]

**Proposition 4** If \( \Gamma \vdash e \vdash (I; O; G) \), then \( G = [\rightarrow e] \).

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

**Corollary 3** If \( \Gamma \vdash e : M \), then \( \vdash \rightarrow e \).

**Lemma 1** If \( \Gamma \vdash e \vdash (I; O; G) \) and \( e \xhookrightarrow{e} e \), then \( e \) is a structure and \( \vdash \rightarrow e \) and \( G = [\rightarrow e] \).

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

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

**Proof** Assume there is a cycle in \( \vDash_o = (\vDash_o \cup (\vDash_o)^+) \). This cycle cannot contain only \( \vDash_o \) edges,
since for all nodes \( N_1, N_2, N_3 \) such that \( N_1 \vDash_o N_2 \vDash_o N_3 \), by definition \( N_1 \rightarrow_o N_2 \rightarrow_o N_3 \), with \( o(N_1) \notin Predictable \) and \( o(N_2) \notin Predictable \), and by definition of \( \rightarrow_o \) and hypothesis 2, we have \( o(N_2) \in Predictable \), which is a contradiction.

So there is at least one \( \rightarrow_o^{\downarrow} \) edge in our cycle. But \( \vDash_o \) is included in \( (\vDash_o)^+ \), so this is a cycle for \( (\vDash_o)^+ \) too. \( \square \)
4.3.3  Manipulation of recursive bindings

**Definition 15 (Graph comparison)**  We define $\rightarrow_1 < \rightarrow_2$ by

- for all $N_1 \overset{\rightarrow_2}{\rightarrow} N_2$, there exists $N_1 \overset{\rightarrow_1}{\rightarrow} N_2$
- for all $N_1 \overset{\rightarrow_2}{\rightarrow} N_2$, there exists $N_1 \overset{\lambda}{\rightarrow} N_2$.

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

**Definition 16 (Binding comparison)** A binding $b_1$ is more restrictive than a binding $b_2$ (written $b_1 < b_2$) iff they have the same domains ($\text{dom}(b_1) = \text{dom}(b_2)$), they define variables in the same order ($\rightarrow_k = \rightarrow_k'$), the dependencies and shapes of $b_1$ are more restrictive than those of $b_2$ ($\rightarrow_k < \rightarrow_k'$, and for all $x \in \text{dom}(b_2)$, if $b_2(x) \notin \text{Predictable}$, then $b_1(x) \notin \text{Predictable}$).

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

**Lemma 3 (Relax)** If $b' < b$ and $\triangleright \triangleright b, then \triangleright b$.

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

- If $y \triangleright \triangleright \text{Output}(b) x$, then $y \overset{\lambda}{\rightarrow_b} x$ and $b(y) \notin \text{Predictable}$. By definition of $b' < b$, this implies that $b'(y) \notin \text{Predictable}$ and $y \overset{\lambda}{\rightarrow_{b'}} x$. Whatever $\chi$ is, it is a right-to-left edge in $\triangleright \triangleright$, which contradicts $\triangleright b'$.

- If $y \overset{\lambda}{\rightarrow_b} x$, by definition of $b' < b$, this implies that $y \overset{\lambda}{\rightarrow_{b'}} x$, so it is a right-to-left edge in $\triangleright \triangleright$, which contradicts $\triangleright b'$.

$\square$

**Lemma 4**  If $\vdash \rightarrow_{(\gamma;\sigma)}$, then $\vdash \text{Bind}(\sigma)$.

**Proof** $\text{Bind}(\sigma)$ is in the same order as $\sigma$, and its graph does not take fake dependencies into account. Lemma 3 allows to conclude. $\square$

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

**Definition 17 (Paths)**  For a path $\delta = [N_0 \overset{\lambda_1}{\rightarrow} \ldots \overset{\lambda_n}{\rightarrow} N_n]$, we define the degree of $\delta$ as $\chi_n$, and we write $\delta^\chi$ for a path of degree $\chi$, and $\delta \subseteq \rightarrow$ if $\delta$ is a path of $\rightarrow$.

Eventually, we write edges as triples (source, target, degree), and paths as lists of paths such that the target of one is the source of the next one, separated by commas, as in $\delta_1^\chi_1, (x, y, \chi), \delta_2^\chi_2$. 

73
Proposition 5 (Let rec internal dependencies)

For all \( y \), for all \( x \in FV(e) \setminus \text{dom}(b) \), \( \text{Degree}(x, \text{let rec } b \text{ in } e) \leq \text{Degree}(x, e) \).

For all \( y \), for all \( x \in FV(b(y)) \setminus \text{dom}(b) \), \( \text{Degree}(x, \text{let rec } b \text{ in } e) \leq \text{Degree}(x, b(y)) \).

Proof

Let \( X \neq Y, b = (x_1 = e_1 \ldots x_n = e_n) \), and \( o = (\text{Output}(b), Y \vartriangleright = e) \).

- For the first point, as \( x \in FV(e) \), there is an edge \( X \xrightarrow{\chi}(X; x; e) Y \), where \( \chi = \text{Degree}(x, e) \).
  By hypothesis 2, \( \text{Degree}(x, \text{let rec } b \text{ in } e) \leq \chi \).

- The second point is similar. Suppose \( y = x_i \) and \( f = b(y) \). There is an edge \( X \xrightarrow{\chi}(X; x; e) X_{x_i} \), where \( \chi = \text{Degree}(x, f) \).
  By hypothesis 2, \( \text{Degree}(x, \text{let rec } b \text{ in } e) \leq \chi \).

\( \square \)

Proposition 6 (Merging nested bindings)

Let \( b = (b_1, x = \text{let rec } b_2 \text{ in } e, b_3), b' = (b_1, b_2, x = e, b_3) \), with \( \vdash \) \( b \) and \( \text{dom}(b_2) \perp \text{dom}(b) \cup FV(b_1, b_3) \).

Let \( \delta \) a path of \( \rightarrow_{b} \), from \( x_1 \) to \( x_2 \), of degree \( \chi \).

1. If \( x_1, x_2 \in \text{dom}(b) \), then \( x_1 \xrightarrow{b, +} x_2 \), with \( \chi' \leq \chi \).
2. If \( x_1 \in \text{dom}(b), x_2 \in \text{dom}(b_2) \), then \( x_1 \xrightarrow{b, +} x_2 \), with \( \chi' \leq \chi \).
3. If \( x_1 \in \text{dom}(b_2), x_2 \in \text{dom}(b) \), then if \( \chi = \emptyset \), then \( x_2 \in \{x\} \cup \text{dom}(b_3) \).
4. If \( x_1, x_2 \in \text{dom}(b_2) \), then either \( \delta \subset \rightarrow_{b_2} \)

or \( x \xrightarrow{b, ^{+}} x \) for some \( \chi' \leq \chi \).

Proof By induction on the length of \( \delta \).

Base \( \delta \) is an edge.

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

Induction step \( \delta \) is of length \( n > 1 \).

1. \( x_1, x_2 \in \text{dom}(b) \).
• If $\delta$ only has nodes in $\text{dom}(b)$, let $(x_3, x_2, \chi)$ be its last edge. By induction hypothesis there is a path $\delta_{1}^{x'}$ from $x_3$ to $x_2$ with $x' \leq \chi$ in $\rightarrow_b$, and a path $\delta_{2}^{x''}$ from $x_1$ to $x_3$, so $\delta_{1}^{x''}, \delta_{2}^{x''} \subset \rightarrow_b$, with degree $x' \leq \chi$.
• Otherwise, let $x_2$ be the last node of $\delta$ in $\text{dom}(b_2)$. The next node is necessarily $x$. Let $x_3$ be the last node of $\delta$ in $\text{dom}(b)$ before $x_5$. Let $x_4$ be the next node. (It is in $\text{dom}(b_2)$.) We have
  
  \[
  \delta = \delta_1^{x_1}, (x_3, x_4, \chi_4), \delta_2^{x_2}, (x_5, x, \chi_5), \delta_3^{x_3}, \]
  
  with $\delta_3^{x_3} \subset \rightarrow_b$. Let now $o = (\text{Output}(b_2), Y) \rightarrow_x e$ and consider the structure $(X \triangleright x_3; o)$. Its concrete dependency graph is $\rightarrow_{(X \triangleright x_3; o)}$ and contains a path $(X, x_4, \chi_4), \delta_2^{x_2}, (x_5, Y, \chi_5)$. So by hypothesis 2, we have $\chi_3' = \text{Degree}(x_3, \text{let } b_2 \text{ in } e) \leq \chi_3$, so there is and edge $x_3 \xrightarrow{x_3'} b_6$.

  Then, applying the induction hypothesis to $\delta_1$ and $\delta_3$ if not empty, we obtain two paths $\delta_1^{x_1}$ and $\delta_3^{x_3}$ of $\rightarrow_b$, and so $\delta_1^{x_1}, (x_3, x, \chi_3'), \delta_3^{x_3}$ is a path of $\rightarrow_b$, with a degree $\chi_3' \leq \chi_3$ if $\delta_3$ is empty, and a degree $\chi_3' \leq \chi_3$ otherwise.

2. $x_1 \in \text{dom}(b), x_2 \in \text{dom}(b_2)$. Let $x_3$ be the last node of $\delta$ in $\text{dom}(b)$, and $x_4$ the next one. $\delta$ is of the shape $\delta_1^{x_1}, (x_3, x_4, \chi_3), \delta_2^{x_2}$, with the nodes of $\delta_2$ in $\text{dom}(b_2)$. $\delta_1$ and $\delta_2$ could be empty. As above, by lemma 5, we have $\chi_3' = \text{Degree}(x_3, \text{let } b_2 \text{ in } e) \leq \text{Degree}(x_3, b_2(x_4)) = \chi_3$, so we have an edge $x_3 \xrightarrow{x_3'} b_6$.

• If $\delta_2$ is empty, $n > 1$, so $\delta_1$ is non-empty, and applying induction hypothesis to $\delta_1$, we obtain $\delta_1^{x_1}$ with same ends, and therefore obtain a path in $\rightarrow_b$ with same ends as $\delta$, and with degree $\chi_3' \leq \chi_3$.
• Otherwise, $\delta_2^{x_2} \subset \rightarrow_b$. Let $X \neq Y$, $i = X \triangleright x_3$, and $o = \text{Output}(b_2), Y \rightarrow_x e$. We obtain a path $(x_3, x_4, \chi_3), \delta_2^{x_2}$ in $\rightarrow_{(i; o)}$ with same ends as $\delta$, and with degree $\chi_3'$. So, if $\delta_1$ is empty, we have in both cases a path from $x_3$ to $x$ in $b$, with degree $\chi_3' \leq \chi_3$. Otherwise, by induction hypothesis, we obtain $\delta_1^{x_1}$ with $\chi_1' \leq \chi_1$, and reason exactly as above.

3. $x_1 \in \text{dom}(b_2), x_2 \in \text{dom}(b)$. Assume $\chi = \emptyset$. The first node of $\delta$ not in $\text{dom}(b_2)$ is necessarily $x$. Let $x_3$ be the node just before it. $\delta$ has the shape $\delta_1^{x_1}, (x_3, x, \chi_3), \delta_2^{x_2}$. If $\delta_2$ is empty, we have $x_2 = x$ which is clearly in $\{x\} \cup \text{dom}(b)$. Otherwise, apply induction hypothesis to obtain a path $\delta_2^{x_2}$ with the same ends as $\delta_2$ and $\chi_2' \leq \chi_2$. But here $\chi = \chi_2 = \emptyset$ so $\chi_2' = \emptyset$. As $G_b \vdash b$, $x_2$ must be defined after $x$ in $b$, so it must be in $\text{dom}(b)$.

4. $x_1, x_2 \in \text{dom}(b_2)$. If all the nodes are in $\text{dom}(b_2)$, then $\delta \subset \rightarrow_b$ directly. Otherwise, the first node not in $\text{dom}(b_2)$ in $\delta$ is necessarily $x$. Let $x_3$ be the node just before it. $\delta$ has to continue after $x$, because it has to go back to a node in $\text{dom}(b_2)$, by hypothesis. Let $x_4$ be the node just after the first occurrence of $x$. $\delta$ has the shape $\delta_1^{x_1}, (x_3, x, \chi_3), (x, x_4, \chi_4), \delta_2^{x_2}$.

• If $\delta_2$ is empty, then as $\text{Degree}(x, b_2(x_4)) = \chi_4$, by lemma 5 there exists an edge $x \xrightarrow{x_4'} b_6$ with $x_4' \leq \chi_4$. But here $\chi_4 = \chi$, so we are in the second case and $x \xrightarrow{x_4'} b_6$ with $x_4' \leq \chi$.
• Otherwise, by induction hypothesis on $(x, x_4, \chi_4), \delta_2^{x_2}$, we obtain a path $\delta_2^{x_2} \subset \rightarrow_b$, from $x$ to $x$ and $x_4' \leq \chi_2$, which means that $x \xrightarrow{x_2'} b_6$, and that is enough.

\[\square\]

**Corollary 4 (Correct internal merge)**

If $b = (b_1, x = \text{let } b_2 \text{ in } e, b_3), \vdash b$, $\vdash b_2$, $\text{dom}(b_2) \perp \text{dom}(b) \cup \text{FV}(b_1, b_3)$, and $b' = b_1, b_2, x = e, b_3$, then $\vdash b'$.
Proof We want to prove that if $x_1 \xrightarrow{\bigoplus} x_2$, then $x_1 \succ x_2$ ($x_1$ is defined before $x_2$ in $b'$).

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

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

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

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

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

4.4 Soundness

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

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

Proof By induction on the typing derivation.

Struct. $e = \langle \ell ; o \rangle$ and $M = \langle I ; O ; G \rangle$. By syntactic correctness, $\text{dom}(\ell) \perp \text{Names}(o)$, so $\text{dom}(I) \perp \text{dom}(O)$. Moreover, the targets of $G$, by construction of $\rightarrow_{\ell;o}$, and $\rightarrow_{\ell;\omega}$, are in $\text{dom}(O)$, and by typing $\vdash G_{\ell;\omega}$, so $\vdash G$. Eventually, $\vdash I$ is given by the typing rule, and $\vdash O$ is obtained by induction hypothesis.

Sum. Assume $e = e_1 + e_2$, $\Gamma \vdash e_1 : \langle I_1 ; O_1 ; G_1 \rangle$, $\Gamma \vdash e_2 : \langle I_2 ; O_2 ; G_2 \rangle$, $\vdash G_1 \cup G_2$, and $M = \langle I_1 \cup I_2 \setminus (O_1 \uplus O_2); (O_1 \uplus O_2); G_1 \cup G_2 \rangle$. By induction hypothesis the types of $e_1$ and $e_2$ are well-formed, so $I_1 \cup I_2$ and $O_1 \uplus O_2$ are as well. By construction, the inputs are disjoint from the outputs, the graph is correct, and its targets are in $\text{dom}(O_1 \uplus O_2)$. 76
**Freeze.** \[ e = e' ! X, \Gamma \vdash e' : \langle I ; O ; G \rangle, \text{ and } M = \langle I ; O ; G ! X \rangle. \] The only difficulty is to show that the targets of \(G!X\) are in \(\text{dom}(O)\), but the ones of \(G\) are by induction hypothesis, so it is the same for the ones of \(G'X \triangleright x\), and therefore for the ones of \(G\).

**Close.** Simple by induction hypothesis.

**Project and delete.** Easy by induction hypothesis. For projection for example, everything is trivial, except maybe that \(\text{Targets}(G|_N) \subset \text{dom}(O|_N)\), but by induction hypothesis \(\text{Targets}(G) \subset \text{dom}(O)\), and as \(\text{Targets}(G|_N) = \text{Targets}(G) \cap N\), we have \(\text{Targets}(G|_N) \subset \text{dom}(O) \cap N = \text{dom}(O|_N)\).

**Show and hide.** Assume \(\Gamma \vdash e : \langle I ; O ; G \rangle\), and by rule T-show,

\[ \Gamma \vdash e;x_1,\ldots,x_n : \langle I ; O |_{x_1,\ldots,x_n} ; G |_{x_1,\ldots,x_n} \rangle. \] By induction hypothesis, \(\langle I ; O ; G \rangle\) is well-formed, so \(\vdash I, \vdash O, \vdash G, \text{dom}(I) \perp \text{dom}(O),\) and \(\text{Targets}(G) \subset \text{dom}(O)\). We can deduce that \(\langle I ; O |_{x_1,\ldots,x_n} ; G |_{x_1,\ldots,x_n} \rangle\) is well-formed, since \(\text{Targets}(G|_{x_1,\ldots,x_n}) \subset \{x_1,\ldots,x_n\}\) and by typing \(\{x_1,\ldots,x_n\} \subset \text{dom}(O)\). The other conditions are easy, and hide is similar.

**Rename.** \(e = e'[r], \Gamma \vdash e' : \langle I ; O ; G \rangle, (\text{cod}(r) \setminus \text{dom}(r)) \perp \text{dom}(I) \cup \text{dom}(O)\) \(1)\), and \(M = \langle I \{r\}; O \{r\}; G \{r\} \rangle\).

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

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

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

\[
\text{dom}(I \{r\}) \cap \text{dom}(O \{r\}) \subset \left( (\text{dom}(I) \setminus \text{dom}(r)) \cap \text{cod}(r) \right) \\
\cup \left( (\text{dom}(O) \setminus \text{dom}(r)) \cap \text{cod}(r) \right) \\
\cup \left( r(\text{dom}(I)) \cap r(\text{dom}(O)) \right).
\]

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

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

\[
\text{Targets}(G \{r\}) \subset r(\text{dom}(O)) \cup (\text{dom}(O) \setminus \text{dom}(r)) = \text{dom}(O \{r\}).
\]

**Split.** Assume \(\Gamma \vdash e : \langle I ; O ; G \rangle\), and by rule T-split,

\[ \Gamma \vdash e;x_1,y : \langle I \sqcup \{X : O(X)\}; O \{X \rightarrow Y\}; G |_{x_1,y} \rangle. \] By induction hypothesis, \(\text{Targets}(G) \subset \text{dom}(O)\). But \(\text{Targets}(G |_{x_1,y}) = \text{Targets}(G \setminus \{X\} \cup \{Y\}) \subset \text{dom}(O)\{X \rightarrow Y\}\). The other conditions are easy.

**Other cases.** Easy.

\[ \square \]

**Lemma 6 (Weakening)** If \(\Gamma \vdash e : M\) and \(\text{dom}(\Gamma') \perp \text{FV}(e)\), then \(\Gamma \{\Gamma'\} \vdash e : M\).

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

Now, typing is preserved by the computational contraction rules.

**Lemma 7 (Subject contraction)** If \(e \rightsquigarrow e'\) and \(\Gamma \vdash e : M\), then \(\Gamma \vdash e' : M\).

**Proof** By case analysis on the contraction step.

77
SUM. Assume $e = \langle t_1; o_1 \rangle + \langle t_2; o_2 \rangle$, and $\Gamma \vdash e : M$. By typing we have $\Gamma \vdash \langle t_1; o_1 \rangle : \langle I_1; O_1; G_1 \rangle$, $\Gamma \vdash \langle t_2; o_2 \rangle : \langle I_2; O_2; G_2 \rangle$, and $M = \{ i ; O ; G \}$, with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$. Let $G = G_1 \cup G_2$, and $\Gamma \vdash G$. We have $e' = \langle t; o \rangle$, where $t = (t_1 \cup t_2) \setminus \text{Input}(o_1, o_2)$, $o = o_1 \odot t_2$, with $\langle t_1; o_1 \rangle \odot \langle t_2; o_2 \rangle$.

By lemma 1, $\Gamma \vdash e'$ and $G = [\rightarrow e']$.

Then we deduce easily that $\Gamma \vdash e' : M$:

- $\text{dom}(I) = \text{dom}(I)$ is trivial.
- We have seen that $\Gamma \vdash \langle t; o \rangle$.
- By typing there exist correct $\Gamma_1$ and $\Gamma_2$ such that $\Gamma \langle I_1 \circ t_1^{-1} \uplus \Gamma_1 \rangle \vdash o_1 : \Gamma_1$ and $\Gamma \langle I_2 \circ t_2^{-1} \uplus \Gamma_2 \rangle \vdash o_2 : \Gamma_2$. So it would be enough to derive $\Gamma' \vdash o : (\Gamma_1 \uplus \Gamma_2)$, where $\Gamma' = \Gamma \langle I \circ t^{-1} \uplus \Gamma_1 \uplus \Gamma_2 \rangle$.

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

Then, $I = I_1 \uplus I_2$, with $I'_1 = I_1 \setminus O_2$ and $I'_2 = I_2 \setminus (I_1 \cup O_1)$ and we obtain $I \circ t^{-1} = (I'_1 \circ t^{-1}) \uplus (I'_2 \circ t^{-1}) = (I'_1 \circ t_1^{-1}) \uplus (I'_2 \circ t_2^{-1})$.

Moreover, with $P = \text{cds}(t_1) \cap \text{Variables}(o_2)$, we have $\Gamma_2 = \Gamma'_{ \uplus \text{cds}(t_1) \cap \text{Names}(o) }$, so $\langle t_2; o_2 \rangle$, for all $x \in P$, there is a name $X \in \text{cds}(t_1) \cap \text{Names}(o)$ such that $(X \circ x) \in t_1 \cap \text{Input}(o_2)$, so $\text{id}^1_P = \text{Input}(o_2) \circ t_1^{-1}$, and therefore

$$\Gamma'_{ \uplus o } = \Gamma_2 \circ (\text{id}^1_P) = \Gamma_2 \circ \text{Input}(o_2) \circ t_1^{-1} = \langle \Gamma_2 \circ \text{Input}(o_2) \rangle \circ t_1^{-1} = O_2 \circ t_1^{-1} = O_2 \circ \text{dom}(t_1) \circ t_1^{-1} = \langle O_2 \cap \text{Input}(o_2) \rangle \circ t_1^{-1}$$

So, we obtain $\Gamma_2 = (O_2 \cap I_1) \circ t_1^{-1} \uplus \Gamma_{2, P}$ and so

$$\Gamma' = \Gamma \langle I_1 \setminus O_2 \rangle \circ t_1^{-1} \uplus (I'_2 \circ t_2^{-1}) \uplus (I_1 \cap O_2) \circ t_1^{-1} \uplus \Gamma_{2, P} \rangle$$

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

LIFT.

Let $e = L \vdash \text{let rec } b \text{ in } e_1$, and $\Gamma \vdash e : M$, $\text{dom}(b) \uplus \text{FV}(L)$, and $e' = \text{let rec } b \text{ in } L[e_1]$. By case on $L$. For example $L = \square + e_2$, we have a derivation of the shape

$$\vdash \Gamma \bimp b \quad \Gamma \bimp \langle \Gamma_b \rangle \bimp b : \Gamma_b \quad \Gamma \bimp \langle \Gamma_b \rangle \bimp e_1 : M_1 \quad \vdash \Gamma \bimp e_2 : M_2 \quad I \uplus O_1 \circ I_2 \uplus O_2 \quad \Gamma \bimp (\text{let rec } b \text{ in } e_1) + e_2 : M$$

where $M_1 = \langle I_1; O_1; G_1 \rangle$, $M_2 = \langle I_2; O_2; G_2 \rangle$, and $M = \langle I_1 \cup I_2 \rangle \setminus (O_1 \cup O_2) \circ I_1 \uplus O_2 : G_1 \cup G_2$.

By hypothesis, $\text{dom}(\Gamma_b) = \text{dom}(b) \uplus \text{FV}(e_2)$, so by lemma 6, we have $\Gamma \langle \Gamma_b \rangle \bimp e_2 : M_2$, and we can reconstruct the derivation as follows:

$$\vdash \Gamma \bimp b \quad \Gamma \bimp \langle \Gamma_b \rangle \bimp b : \Gamma_b \quad \Gamma \bimp G_1 \cup G_2 \quad \Gamma \bimp \langle \Gamma_b \rangle \bimp e_1 : M_1 \quad \Gamma \bimp \langle \Gamma_b \rangle \bimp e_2 : M_2 \quad \Gamma \bimp e_1 + e_2 : M$$

$$\vdash \Gamma \bimp \text{let rec } b \text{ in } e_1 + e_2 : M$$
• **FREEZE.** Assume \( e = \langle t; o \rangle ! X \) and \( e' = \langle t; o' \rangle \), with \( o = \langle o_1, X[y^*] : x = f.o_2 \rangle \), and \( o' = \langle o_1, [y^*] : x = f.o_2, X \triangleright y = x \rangle \), with a fresh \( y^* \).

By typing, we have a derivation of the shape

\[
\begin{array}{c}
+ I \quad + \Gamma_c \\
\text{dom}(t) = \text{dom}(I) \\
\forall z \in \text{Variables}(o), \Gamma(I \circ t \downarrow \uplus \Gamma_c) \vdash o(z) : \Gamma_c(z) \\
\hline
\Gamma \vdash \langle t; o \rangle : \langle I; O \uplus G \rangle \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{with } O = \Gamma_c \circ \text{Input}(o) \text{ and } G = [\rightarrow \langle t; o \rangle].
\end{array}
\]

Let \( \Gamma' = \Gamma_c(y \rightarrow \Gamma_c(x)) \). By weakening, we can derive

\[
\forall z \in \text{Variables}(o') \setminus \{ x, y \}, \Gamma(I \circ t \downarrow \uplus \Gamma'_c) \vdash o'(z) : \Gamma'_c(z).
\]

For \( x \) and \( y \), we easily derive too that

\[
\Gamma(I \circ t \downarrow \uplus \Gamma'_c) \vdash f : \Gamma'_c(x) \\
\Gamma(I \circ t \downarrow \uplus \Gamma'_c) \vdash x : \Gamma'_c(y).
\]

So we have

\[
\forall z \in \text{Variables}(o'), \Gamma(I \circ t \downarrow \uplus \Gamma'_c) \vdash o'(z) : \Gamma'_c(z).
\]

Moreover, by lemma 1, we have \( \vdash \rightarrow o' \) and \( [\rightarrow o'] = [G ! X] \), so we can derive

\[
\begin{array}{c}
+ I \quad + \Gamma'_c \\
\text{dom}(t) = \text{dom}(I) \\
\forall z \in \text{Variables}(o'), \Gamma(I \circ t \downarrow \uplus \Gamma'_c) \vdash o'(z) : \Gamma'_c(z) \\
\hline
\Gamma \vdash \langle t; o' \rangle : \langle I; O \uplus [G ! X] \rangle \\
\end{array}
\]

• **DELETE.** Let \( e = \langle t; o \rangle \mid X_1 \ldots X_n \), with \( N = \{ X_1 \ldots X_n \} \), we have

\[
\begin{array}{c}
\text{dom}(t) = \text{dom}(I) \\
+ I \quad + \Gamma_c \\
+ \rightarrow \langle t; o \rangle \quad \Gamma(I \circ t \downarrow \uplus \Gamma_c) \vdash o : \Gamma_c \\
\hline
\Gamma \vdash \langle t; o \rangle : \langle I; O \uplus G \rangle \\
\end{array}
\]

with \( M = (I'; O'; G'; I \uplus O_{\neg N} \uplus O'_{\neg \neg N} ; G_{\neg \neg N}) \) and \( G = [\rightarrow \langle t; o \rangle] \). But necessarily, we have \( e' = \langle t'; o' \rangle = \langle t, \text{Input}(o)_{\neg N} : o_{\neg N} \rangle \).

So, \( I \circ t \downarrow \uplus \Gamma_c = ((I \uplus O_{\neg N}) \circ (t, \text{Input}(o)_{\neg N})^{-1}) \uplus \Gamma'_c \), with \( \Gamma'_c = \Gamma_{\neg N} \), and so \( \Gamma \langle ((I \uplus O_{\neg N}) \circ (t, \text{Input}(o)_{\neg N})^{-1}) \uplus \Gamma'_c \rangle : o_{\neg N} : \Gamma'_c \).

Moreover, we have by lemma 1, \( \vdash \rightarrow o' \) and \( G' = [\rightarrow o'] \), so we can derive

\[
\begin{array}{c}
\text{dom}(t') = \text{dom}(I') \\
+ I' \quad + \Gamma'_c \\
+ \rightarrow \langle t; o' \rangle \quad \Gamma(I' \circ o \downarrow \uplus \Gamma'_c) \vdash o' : \Gamma'_c \\
\hline
\Gamma \vdash e' : \langle I'; O'; G' \rangle \\
\end{array}
\]

• **PROJECT.** Let \( e = \langle t; o \rangle \mid X_1 \ldots X_n \). Let \( N = \{ X_1 \ldots X_n \}, N' = \text{Names}(o) \setminus N, \text{ and } e'' = \langle t; o \rangle \mid N' \). We have in fact that \( e'' \sim e'' \), because of the duality of delete and project.

So if we show that \( \Gamma \vdash e'' : M \), we can reproduce exactly the delete case as above.

By typing, we have \( \Gamma \vdash \langle t; o \rangle : \langle I; O \uplus G \rangle \), and \( M = (I'; O'; G'; I \uplus O_{\neg N}, O' = O_{\neg N}, \text{ and } G' = G_{\neg \neg N} \).

But we can derive \( \Gamma \vdash e'' : (I''; O''; G''); \) with \( I'' = I \cup O_{\neg N} = I \cup O_{\neg \neg N} = I' \), \( O'' = O_{\neg N} \), and \( G'' = G_{\neg N} = G_{\neg \neg N} = G' \), so we derive \( \Gamma \vdash e'' : M \), and may apply the same process as above to deduce \( \Gamma \vdash e' : M \).
• **RENAME.** Let \( e = \langle i; o \rangle [r] \). We have \( e' = \langle i \{ r \}; o \{ r \} \rangle \), and by typing:

\[
\begin{align*}
\vdash I & \quad \vdash \Gamma_o \\
\vdash \rightarrow_{\langle i; o \rangle} & \quad \text{dom}(I) = \text{dom}(i) \\
\Gamma & \vdash \{ i; o \} : \langle I; O; G \rangle \\
\langle \text{cod}(r) \setminus \text{dom}(r) \rangle & \perp \langle \text{dom}(I) \cup \text{dom}(O) \rangle \\
\Gamma & \vdash e : M 
\end{align*}
\]

with \( M = \langle I' ; O'; G' \rangle = \langle I \{ r \}; O \{ r \}; G \{ r \} \rangle \), \( G = [\rightarrow_{\langle i; o \rangle}] \), and \( O = \Gamma_o \circ \text{Input}(o) \).

We may write \( e' \) as \( \langle i'; o' \rangle = \langle i \{ r \}; o \{ r \} \rangle \), and \( \text{Input}(o') = \text{Input}(o) \circ r_{\text{Names}(o)}^{-1} \), so

\[
\begin{align*}
\Gamma_o \circ \text{Input}(o') & = \Gamma_o \circ \text{Input}(o) \circ r_{\text{Names}(o)}^{-1} \\
& = O \circ r_{\text{Names}(o)}^{-1} \\
& = O \circ r_{\text{dom}(O)} \\
& = O'.
\end{align*}
\]

For inputs, we have \( I' \circ I^{-1} = I \{ r \} \circ (i \{ r \})^{-1} = I \circ r_{\text{dom}(I)}^{-1} \circ r_{\text{dom}(i)}^{-1} = I \circ I^{-1} \), so

\[
\Gamma(I' \circ I^{-1} \cup \Gamma_o) \vdash o' : \Gamma_o.
\]

Moreover, it is easily seen that \( \text{dom}(I') = \text{dom}(i') \), \( \vdash I' \), and by lemma 1, we have, with \( \vdash \rightarrow_{\langle i'; o' \rangle} \) and \( G' = [\rightarrow_{\langle i'; o' \rangle}] \), so we can derive

\[
\begin{align*}
\vdash I' & \quad \Gamma_o \\
\vdash \rightarrow_{\langle i'; o' \rangle} & \quad \text{dom}(i') = \text{dom}(I') \\
\Gamma & \vdash \{ i'; o' \} : \langle I'; O'; G' \rangle \\
\end{align*}
\]

• **CLOSE.** Let \( e = \text{close}(e; o) \). We have \( e' = \text{let rec } \text{Bind}(\mathcal{F}) \text{ in } \text{Record}(o) \), and \( \vdash \text{Bind}(\mathcal{F}) \), and by typing

\[
\begin{align*}
\vdash & \quad \rightarrow_{\langle i; o \rangle} \\
\Gamma_o & \vdash o : \Gamma_o \\
\Gamma & \vdash e : M 
\end{align*}
\]

with \( M = \{ O \} \), \( G = [\rightarrow_{\langle i; o \rangle}] \), and \( O = \Gamma_o \circ \text{Input}(o) \).

Let

\[
\begin{align*}
o & = d_1, \ldots, d_n \\
d_i & = \text{L}_i[x_1, \ldots, x_n] \triangleright x_i = e_i \\
b & = \text{Bind}(o) = (x_1 = e_1, \ldots, x_n = e_n) \\
s & = \text{Record}(o) = (X_1 = x_{\mu(1)}, \ldots, X_m = x_{\mu(m)}) \\
\end{align*}
\]

where \( \mu : \{ 1 \ldots m \} \rightarrow \{ 1 \ldots n \} \) injective

and for all \( i \), \( X_i = L_{\mu(i)} \).

We have \( e' = \text{let rec } b \text{ in } s \) and let \( \Gamma_o = \{ x_i : M_i \mid i \in \{ 1 \ldots n \} \} \).

We have

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

so it is ok.

80
• Show. Assume \( e = e_0 : X_1 \ldots X_n \), with \( e_0 = \langle \ell ; o \rangle \). Then, \( e' = \langle \ell ; o' \rangle \), and \( o' = \text{Show}(o, X_1 \ldots X_n) \). Let \( \mathcal{N} = \{ X_1 \ldots X_n \} \). The typing derivation is of the shape

\[
\begin{array}{c}
\vdash I \\
\vdash \Gamma_o
\end{array}
\quad
\frac{\vdash \langle \ell ; o \rangle \quad \text{dom}(I) = \text{dom}(\ell) \quad \Gamma \vdash o : \Gamma_o}{\Gamma \vdash e_0 : \langle I ; O ; G \rangle} \quad \text{dom}(O) \subseteq \text{dom}(G)
\]

with \( G = [\langle \ell ; o \rangle], \ G' = [G_{\mathcal{N}}], \) and \( O = \Gamma_o \circ \text{Input}(o) \).

By lemma 1, we have \( \vdash o' \) and \( G' = [G_{\mathcal{N}}] \).

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

• Hide. As for delete and project, we obtain the expected result by reasoning dually to the Show case.

• Split. Let \( e_0 = \langle \ell ; o \rangle \) and \( e = e_0 : X = Y, \) with \( o = \langle o_1, X[z^*] \triangleright x = e_1, o_2 \rangle \). We have \( e' = \langle \ell ; o' \rangle = \langle \ell, X \triangleright x; o_1, X[z^*] \triangleright y = e_1, o_2 \rangle \) for a fresh \( y \).

The typing derivation is of the shape

\[
\begin{array}{c}
\vdash I \\
\vdash \Gamma_o
\end{array}
\quad
\frac{\vdash \langle \ell ; o \rangle \quad \text{dom}(I) = \text{dom}(\ell) \quad \Gamma \vdash o : \Gamma_o}{\Gamma \vdash e : \langle I ; O ; G \rangle} \quad Y \notin \text{dom}(O) \cup \text{dom}(I)
\]

with \( G = [\langle \ell ; o \rangle], \) and \( O = \Gamma_o \circ \text{Input}(o) \).

By lemma 1, we have \( \vdash o' \) and \( [\langle \ell ; o' \rangle] = [G_X \triangleright Y] \).

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

\( \square \)

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

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

**Lemma 8 (Lift context)** If \( \Gamma \vdash L[e] : M, \ \Gamma \vdash e : M', \) and \( \Gamma \vdash e' : M', \) then \( \Gamma \vdash L[e'] : M. \)

**Proof** By case on \( L. \)

- \( L = \{ \mathcal{S} \} \), with \( \mathcal{S} = s_e, X = \square, s. \) We have a derivation of the form

\[
\begin{array}{c}
\forall (Y = f) \in (s_e, s), \Gamma \vdash f : O(Y) \\
\Gamma \vdash e : M'
\end{array}
\]

with \( M = \{ O \cup \{ X : M' \} \} \).

By hypothesis we have \( \Gamma \vdash e' : M' \), so we can reconstruct the derivation

\[
\begin{array}{c}
\forall (Y = f) \in (s_e, s), \Gamma \vdash f : O(Y) \\
\Gamma \vdash e' : M'
\end{array}
\]

\( \Gamma \vdash \mathcal{S}[e] : M \)
• $L = op[\Box]$, for $op \in \{\text{close}, [r], !X, !x_1...x_n \}$, We have a derivation of the shape

$$
\Gamma \vdash e : M' \quad \text{side conditions} \quad \frac{\Gamma \vdash e : op[M']}{
\Gamma \vdash e : op[M']}
$$

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

• $L = \Box + e_1$. The derivation is of the form

$$
\frac{\Gamma \vdash e + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \cup O_2; G_1 \cup G_2 \rangle}{\Gamma \vdash e : (I_1; O_1; G_1) \quad \Gamma \vdash e_2 : (I_2; O_2; G_2) \quad \Gamma \vdash e : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \cup O_2; G_1 \cup G_2 \rangle}
$$

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

• $L = v + \Box$. Similar.

□

This property is true for multiple lift contexts as well.

**Lemma 9 (Multiple lift context)** If $\Gamma \vdash F[e] : M$, $\Gamma \vdash e : M'$, and $\Gamma \vdash e' : M'$, then $\Gamma \vdash F[e'] : M$.

**Proof** By trivial induction on $F$. □

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

**Proof** Trivial. □

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

**Lemma 10 (Evaluation context)** Assume $\Gamma \vdash E[e] : M$, with a sub-derivation $\Gamma' \vdash e : M'$ in place of the hole. Assume also that $\Gamma(\Gamma') \vdash e' : M'$, that $e \in \text{Predictable}$ and $e' \in \text{Predictable}$, and that for all $x \in \text{FV}(e')$, $x \in \text{FV}(e)$ and $\text{Degree}(x,e) \leq \text{Degree}(x,e')$.

Then $\Gamma \vdash E[e'] : M$.

**Proof** By induction on $E$.

• $E = F$. By lemma 9.

• $E = \text{let rec } b_e \text{ in } F$. The derivation has shape

$$
\vdash b_e \quad \vdash \Gamma(\Gamma_{b_e}) \vdash b_e : \Gamma_{b_e} \quad \vdash \Gamma(\Gamma_{b_e}) \vdash F[e] : M
$$

By lemma 9, we have $\Gamma \vdash F[e'] : M$, so we can reconstruct the derivation compositionally.

82
\[ E = \text{let rec } \mathbb{B} \sqcup [e] \text{ in } f, \text{ with } \mathbb{B} = b_v, x = \square, b. \] The derivation has the shape

\[
\frac{\forall y \neq x \in \text{dom}(\mathbb{B}), \Gamma(\Gamma_b) \vdash \mathbb{B}(y) : \Gamma_b(y) \quad \Gamma(\Gamma_b) \vdash \mathbb{F}[e] : \Gamma_b(x) \quad \vdots}{\Gamma \vdash E[e] : M} \quad \frac{\vdash b}{\Gamma \vdash f : M \vdash b}
\]

where \(b = \mathbb{B} \sqcup [e]\).

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

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

As \(\vdash b\), we have \(\succ_b \vdash b'\), since they define the same variables in the same order.

Obviously, we have \(\succ_b = \succ_{b'}\).

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

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

\(\Box\)

**Lemma 11 (Evaluation context)** If \(e \sim_c e'\), and \(\Gamma \vdash E[e] : M\), then \(\Gamma \vdash E[e'] : M\).

**Proof** By lemma 10. \(\Box\)

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

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

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

**Proof** Assume \(\vdash b\). Then, \(b\) and \(b'\) define the same variables in the same order. So, \(\succ_b \vdash b'\).

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

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

For this we remark that

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

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

- For variables \(z \in \text{FV}(v) \setminus \text{Capt}(\mathbb{N}[x])\), we have \(\chi = \emptyset\).
• For variables $z \not\in FV(v) \setminus \text{Capt}_\alpha(\mathbb{F}[\mathbb{N}])$, we have $\chi = \text{Degree}(z, \mathbb{F}[\mathbb{N}])$.

Therefore, we have

$\rightarrow \nu \subset \rightarrow_h$

$\cup \{ z \xrightarrow{\nu} y \mid z \in FV(v) \setminus \text{Capt}_\alpha(\mathbb{F}[\mathbb{N}]) \}$

$\cup \{ z \xrightarrow{\Delta} y \mid z \in FV(\mathbb{F}[\mathbb{N}[v]]) \cap (FV(v) \setminus \text{Capt}_\alpha(\mathbb{F}[\mathbb{N}])), \chi = \text{Degree}(z, \mathbb{F}[\mathbb{N}]) \}$

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

For each edge in $\{ z \xrightarrow{\nu} y \mid z \in FV(v) \setminus \text{Capt}_\alpha(\mathbb{F}[\mathbb{N}]) \}$, as $z \in FV(v)$, there is an edge $z \xrightarrow{\Delta} x$. But by hypothesis 2, $\text{Degree}(x, \mathbb{E}[\mathbb{N}[x]]) = \emptyset$, so there is a strict path from $z$ to $y$ in $\rightarrow_h$.

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

So, we have $\rightarrow_h \subset \rightarrow''$, and by transitivity of graph comparison, we get $\rightarrow_h \subset \rightarrow \nu$.

$\Box$

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

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

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

• $\mathbb{E} = \mathbb{F}$, impossible.

• $\mathbb{E} = \text{let rec } b \in \mathbb{F}$. By corollary 5.

• $\mathbb{E} = \text{let rec } b \in e$. Let $b = \mathbb{B}[\mathbb{F}[\mathbb{N}[x]], b' = \mathbb{B}[\mathbb{F}[\mathbb{N}[v]]]$, and $\mathbb{B} = \mathbb{b}, y = \Box, b_1$.

The derivation has the shape

\[
\vdash b \quad \forall y \in \text{dom}(b, b_1), \Gamma(\Gamma_b) \vdash \mathbb{B}(y) : \Gamma(y) \\
\hline
\Gamma \vdash \mathbb{E}[\mathbb{N}[x]] : M
\]

We have $b_1(x) = v$, and by lemma 12, $\vdash b'$.

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

\[
\vdash b' \quad \forall y \in \text{dom}(b'), \Gamma(\Gamma_b) \vdash b' : \Gamma_b(y) \\
\hline
\Gamma \vdash \mathbb{E}[\mathbb{N}[v]] : M
\]

$\Box$

Type preservation along the IM rule is proven.

**Lemma 14 (Internal merge)**

If $e = \text{let rec } b_1, x = (\text{let rec } b_1 \in e_1), b_2 \in e'$, and $\Gamma \vdash e : M$, then $\Gamma \vdash e' : M$. 

84
Proof

We have a derivation of the shape

$$
\forall y \neq x \quad \vdash \Gamma_b(y) : \Gamma_b(y)
$$

\[\vdash \Gamma_b, b_1 : \Gamma_1 \vdash b_1 : \Gamma_b \vdash b : \Gamma_b \vdash e_1 : \Gamma_b(x) \vdash \Gamma_b \vdash \text{let rec } b_1 \text{ in } e_1 : \Gamma_b(x) \vdash \Gamma_b \vdash \text{let rec } b \text{ in } f : \Gamma_b(\Gamma_b) \vdash f : M \vdash b \]

where \( b = b_v, x = \text{(let rec } b_1 \text{ in } e_1, b_2 \text{ in } f \text{ in } M \)

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

Moreover, by weakening, we have

$$
\forall y \neq x \quad \vdash \Gamma_b(y) : \Gamma_b(y)
$$

and with \( \Gamma_{b'} = \Gamma_b \cup \Gamma_{b_1} \),

$$
\vdash \Gamma_{b'} : f : M
$$

and we have

$$
\forall y \in \text{dom}(b') \quad \vdash \Gamma_{b'}(y) : \Gamma_{b'}(y) \quad \vdash \Gamma_{b'} : f : M \vdash b' \quad \vdash \text{let rec } b' \text{ in } f : M
$$

\( \square \)

Next, rule EM is examined.

Lemma 15 (External merge)

If \( \text{dom}(b) \perp (\text{dom}(b_v) \cup \text{FV}(b_v)) \), then \( \vdash e_0 : M \).

\( e_0 = \text{let rec } b_v \text{ in } \text{let rec } b \text{ in } e, \)

\( e'_0 = \text{let rec } b_v, b \text{ in } e, \text{and} \)

\( \Gamma \vdash e_0 : M, \)

Proof The typing derivation for \( e_0 \) has the shape

\[\vdash b_v \quad \vdash b \quad \vdash \Gamma_1 \vdash b_1 : \Gamma_1 \quad \vdash \Gamma_2 \vdash b_2 : \Gamma_2 \quad \vdash \Gamma(\Gamma_1) \vdash e : \Gamma_2 \quad \vdash \Gamma(\Gamma_1) \vdash \text{let rec } b \text{ in } e : \Gamma_1 \quad \vdash e_0 : M \]

By lemma 5, we have \( \vdash b_v, b \).

So by weakening we can reconstruct the derivation.

\( \square \)

We can now state the subject reduction property.
Lemma 16 (Subject reduction) If $e \rightarrow e'$ and $\Gamma \vdash e : M$, then $\Gamma \vdash e' : M$.

Proof By immediate induction, with lemmas 7, 11, 13, and 15. □

Eventually, we prove that if a term is well-typed and is not a result, then either it reduces to another term, or it is stuck on a free variable. This is known as the progress property.

Lemma 17 (Progress) If $\Gamma \vdash e : M$ and $e$ is not a result, then either $e = E[N[x]]$ with $x \not\in \text{Capt}_n(E[N])$, or there exists $e'$ such that $e \rightarrow e'$.

Proof By induction on $e$.

1. If $e$ is of the shape $L[e_0]$, and $e_0$ is not a value. If $e_0 = \text{let rec } b \text{ in } f$, then the Lift applies.

   Else, by induction hypothesis we are in one of the following cases.
   
   - $e_0 = E[N[x]]$ with $x \not\in \text{Capt}_n(E[N])$, and $e$ is stuck on $x$ too, i.e. $e = L[E[N[x]]]$.
   - Otherwise, if $e_0 \rightarrow e'_0$, we reason by case analysis on the applied reduction rule.
     
     - $\text{EM}$. Then the Lift rule applies for $e$.
     
     - $\text{SUBST or CONTEXT}$. Then $e_0 = E[f]$ and $e'_0 = E[f']$. By case analysis again, on $E$:
       
       * If $E = \square$ or $E = F$, then $e$ reduces by the same rule, since $L[E]$ is an evaluation context.
       
       * If $E = \text{let rec } b_\nu \text{ in } F_0$ or $E = \text{let rec } B[F]$ in $g$, then the Lift rule applies for $e$.

2. If $e$ is of the shape $N[x]$, there is nothing to show ($x$ is necessarily free in $N[x]$).

3. $e = \text{let rec } b \text{ in } f$.

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

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

   - $f \rightarrow f'$. By case analysis on the reduction:

     - $\text{EM}$. Then rule EM applies for $e$ as well.

     - $\text{SUBST or CONTEXT}$.

       We have $f = E[g]$ and $f' = E[g']$. If $E = \text{let rec } b_\nu' \text{ in } F'$ or $E = \text{let rec } B[F]$ in $g$, then rule EM applies, and otherwise, the same rule applies for $e$ since let rec $b_\nu$ in $E$ is an evaluation context.

     - $f = E[N[x]]$, with $x \not\in \text{Capt}_n(E[N])$.

       If $E = \text{let rec } b_{\nu}' \text{ in } F$ or $E = \text{let rec } B[F]$ in $g$, then rule EM applies, and otherwise, $E$ is of the shape $F$ and $f = F[N[x]]$, so $\text{let rec } b_\nu \text{ in } E$ in $F[N[x]]$. If $x \in \text{dom}(b_\nu)$, then rule SUBST applies, and otherwise $e = E_0[N[x]]$ with $x \not\in \text{Capt}_n(E_0)$.

   (b) Otherwise, $b$ is not evaluated, so $b$ is of the shape $b_{\nu}, y = g, b_1$, where $g$ is not a value.

   - If $g$ is a result, then it is of the shape $\text{let rec } b_{\nu}' \text{ in } v$, and by internal merge, $e \rightarrow \text{let rec } b_{\nu}, b_{\nu}', y = v, b_1 \text{ in } f$.

   - Otherwise, by induction hypothesis:

     - If $g \rightarrow g'$, by case on the reduction.

      * $\text{EM}$: then rule IM applies for $e$.

      * $\text{CONTEXT or SUBST}$: then $g = E[g_\kappa]$ and $g' = E[g'_\kappa]$. If $E$ is of the shape $\text{let rec } b_{\nu}' \text{ in } F$ or let rec $B[F]$ in $g''$, then rule IM applies for $e$, and otherwise, the global context is an evaluation context and the same rule $\text{CONTEXT}$ or $\text{SUBST}$ applies for $e$.  

86
- If \( g = E[N[x]] \) with \( x \notin \text{Capt}_\Gamma[E[N]] \). By case on \( E \). First notice that we know that \( x \notin \text{dom}(y = g, b_1) \), since by typing \( \vdash b \) and therefore if \( x \rightarrow_b^+ y \), then \( x \) is defined before \( y \) in \( b \), and \( g = E[N[x]] \) implies the existence of an edge \( x \rightarrow_b y \) by hypothesis 2.
  * If \( E = \text{let rec } b' \text{ in } F \) or \( \text{let rec } E\ [E] \text{ in } g' \), then rule IM applies.
  * Else, if \( x \in \text{dom}(b_v) \), then rule Subst applies, since the global context is an evaluation context.
  * Else, if \( x \notin \text{dom}(b_v, y = g, b_1) \), then \( e \) is of the shape \( E_0[N[x]] \) with \( x \notin \text{Capt}_\Gamma(E_0) \).

4. \( e = e_1 + e_2 \).

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

\[
\frac{\vdash \langle \ell_1 ; o_1 \rangle \quad \vdash I_1 \quad I_1 \triangleright o_1 \vdash \Gamma_1 \quad \vdash \langle \ell_2 ; o_2 \rangle \quad \vdash I_2 \quad I_2 \triangleright o_2 \vdash \Gamma_2}{\vdash e : \langle \ell_1 ; o_1 ; \Gamma_1 \rangle + \langle \ell_2 ; o_2 ; \Gamma_2 \rangle}
\]

with \( \vdash G_1 \cup G_2 \) and \( I_1 \triangleright O_1 \triangleright I_2 \triangleright O_2 \) as side-conditions and

\[
\begin{align*}
G_1 &= \langle \ell_1 ; o_1 \rangle \\
G_2 &= \langle \ell_2 ; o_2 \rangle \\
O_1 &= \Gamma_1 \circ \text{Input}(o_1) \\
O_2 &= \Gamma_2 \circ \text{Input}(o_2)
\end{align*}
\]

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

So we may assume that \( e_1 = \langle \ell_1 ; o_1 \rangle \), \( e_2 = \langle \ell_2 ; o_2 \rangle \), and that bound variables of the two structures meet only on the common names, i.e. \( e_1 \equiv e_2 \). This can be reached via \( \alpha \)-conversion.

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

5. \( \text{Close. } e = \text{close } e_0 \), and \( e_0 \) is a value, not a variable (these cases have been treated above).

By typing, \( e_0 = \langle e ; o \rangle \) and we have

\[
\frac{\vdash e_0 \quad \vdash \langle e ; o \rangle \quad \vdash \Gamma \circ e_0 \quad \vdash o : \Gamma_o}{\vdash e_0 : \langle e ; o \rangle}
\]

So we have \( e \rightarrow \text{let rec } \text{Bind}(e) \text{ in } \text{Record}(o) \), provided \( e \) is defined and \( \text{Bind}(e) \) is syntactically correct.

By lemma 2, \( e \) is defined and \( \vdash \text{Bind}(e) \).

For any forward reference from \( x \) to \( y \) in \( \text{Bind}(e) \), there is an edge from \( y \) to \( x \) in \( e \), and if it points to a component of unpredictable shape, then either its degree is \( \emptyset \) or we have \( y \triangleright_o x \), so \( y \) is defined before \( x \) in \( e \) and therefore, in both cases, \( x \) is defined before \( y \) in \( \text{Bind}(e) \).

6. Other operators trivial.
Eventually, we can prove a standard soundness theorem.

**Theorem 2 (Soundness)** The evaluation of a well-typed expression may either not terminate, or reach a result, or get stuck on a free variable.

As free variables cannot appear during reduction, we have the following more standard corollary.

**Corollary 6 (Soundness)** The evaluation of a closed well-typed expression may either not terminate or reach a result.
Chapter 5

Refined static semantics: type components

In this chapter, we extend the MML language with type definitions and abstract types. Our formalization is strongly inspired by Leroy’s module systems \([51, 52, 53, 54]\), but the theoretical design also bases on type theory for recursive modules \([27, 29]\), and the work of Duggan and Sourelis \([31]\) and Flatt and Felleisen \([36, 35]\). It does not solve any of the issues related to these systems, such as undecidability, lack of principal signatures, or even problems for syntactically represent signatures \([56, 65]\). It should rather be seen as an experiment on the expressive power of mixin modules with type components.

5.1 The MML language

Our language of mixin modules with type components is presented in figure 5.1. Names \(S \in \text{Names}\) are distinguished from \(S \in \text{V}ars\). Expression variables \(x \in \text{MVars} \subseteq \text{Vars}\) are distinguished from \(x \in \text{TVars} \subseteq \text{Vars}\). Expression names \(X \in \text{MNames} \subseteq \text{Names}\) are distinguished from \(X \in \text{TNames} \subseteq \text{Names}\). Variables are used as binders, as usual. Names are used for accessing to definitions in mixin modules, as an external interface to other parts of the expression. A label \(L\) can be either a name or the anonymous label \(\_\).

The syntax comprises two main syntactic classes, expressions, which represent computational instructions, and types, which contain static information, roughly.

**Definitions** Expressions build on definitions \(d\), and outputs \(o = (o_1 \ldots o_n)\), which are lists of definitions. A definition \(d\) can have two shapes. If \(d = (T \triangleright t = M)\), it binds a type expression \(M\) to both a type name \(T\) and a type variable \(t\). It is then called a *type definition*. If \(d = (L \triangleright x[y^*] = e)\), then it binds an expression \(e\) (the *body* of the definition) and a finite set of variables \(y^*\) to a label \(L\) and an expression variable. It is then called a *value definition*. The name \(X\) or \(T\) is used by other parts of the program to refer to the bound object. Conversely, the variable \(x\) or \(t\) is used by other definitions under the scope of the definition to refer to the bound object. If the label is the anonymous label, the bound object remains unaccessible to other parts of the program. The attached set of variables represents fake dependencies that help the programmer specify the order of evaluation, exactly as for MM in chapter 3: when a mixin module is instantiated to a module, by the close operator, definitions are reordered, taking actual and fake dependencies into account.

**Module expressions** Intuitively, expressions are divided into three parts. A basic module \(\{a_v\}\) is a list \(a_v\) of pairs \(S \triangleright s\) of a name and a variable. For homogeneity, we consider these pairs as
<table>
<thead>
<tr>
<th>Variable</th>
<th>Expression</th>
<th>Type</th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>t</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>T</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>L ∈ Names ⊔ {}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Path: \( p ::= x \mid p.X \)

Expression: \( e ::= p \) \hspace{1cm} \text{Path}
| \{o_c\} | Module |
| \text{let rec } o \text{ in } e | \text{let rec} |
| \langle I; o \rangle | Structure |
| \( e_1 + e_2 \mid \text{close } e \) | Composition, closure |
| \( e : M \) | Type constraint |

Output: \( o ::= d_1 \ldots d_n \)
Definition: \( d ::= L \triangleright x[y^\ast] = e \mid T \triangleright t = M \)

Type: \( M ::= * \) \hspace{1cm} \text{New type}
| \( t \mid p T \) | Type path |
| \langle O \rangle | Module type |
| \langle I; O ; \rightarrow ; \rightarrow \rangle | Mixin module type |

\( I, O ::= D_1 \ldots D_n \) \hspace{1cm} \text{Signature}
\( D ::= L \triangleright s : M \) \hspace{1cm} \text{Declaration}
\( \rightarrow \mathcal{C}_\text{fin} \{ X \triangleright_{\ast} Y \mid X, Y \in \text{Name}, \chi \in \text{Degrees} \} \) \hspace{1cm} \text{Dynamic graph}
\( \rightarrow \mathcal{C}_\text{fin} \text{Names} \times \text{Name} \) \hspace{1cm} \text{Static graph}

\( \text{Degrees} = \{ \emptyset, \varnothing \} \)

**Figure 5.1: Syntax of MM**

definitions, such that the body of each value definition is a variable, not bound by the current module. This way, basic module expressions are always values. Modules are required not to bind the same name or the same variable twice. Moreover, because there is no reordering on module definitions, fake dependencies do not make any sense so we do not write them. Module selection is performed by the selection operator, but it is restricted to paths \( p = x . X_1 . . . . X_n \). The rationale for restricting selection to paths has to do with phase distinction [41]. Roughly, by avoiding computational expressions in types, we avoid fully dependent types, and the associated difficulties, such as the undecidability of type equivalence.

**Mixin module expressions** Basic mixin modules, called structures, are pairs \( \langle I; o \rangle \) of an input \( I \) (also called a signature) and an output \( o \). An input is a finite set \( I = (D_1 \ldots D_n) \) of declarations. A declaration \( D \) is the specification of a definition, that the mixin module expects as an input. It can either be of the shape \( X \triangleright x : M \), and give the type of a value definition, or give the type of a type definition. The point is that a type definition \( T \triangleright t = M \) can be a concrete (or manifest) one \( T \triangleright t : M \), but can also be abstracted over. The type \( t \) provided as an input to the mixin module can then be any type. The corresponding declaration is \( T \triangleright t : * \). We call it an abstract declaration. The scope of binding variables in \( I \) is the whole structure, whereas the scope of the binding variables in \( o \) is restricted to \( o \). The input is required not to bind the same name or the same variable twice, as well as the output. Moreover, although the input and the output are allowed to define some names in common, they must not define variables in common. Fake dependencies in definitions are requested to refer to variables defined in the same structure. Mixin module expressions come with a minimal set of operators: composition \( + \) and instantiation \( close \). Other usual mixin module operators are left for the moment, since they would complicate the presentation. They are used in examples in section 5.4.
**Types** Type expressions in MML include the unknown type $\star$, type variables $t$, access to type definitions in modules $\mu T$, module types $\{O\}$, and mixin module types $\langle I; O; \to; \to \rangle$. Both $I$ and $O$ range over finite sets of declarations. They are called the input and output signatures, respectively. In module or mixin module types, signatures should not define the same name twice, and not define the same variable twice either. In mixin module types, $I$ and $O$ should not define any variable in common, but are allowed to define some names in common. In module types, abstract declarations make the implementation of the declared type hidden to outer parts of the program. In mixin module types, an abstract input declaration does not have the same meaning, it specifies that no constraint is put on the input type; any type is accepted. The graph $\rightarrow$ is a finite graph over expression names, labeled by degrees $\chi \in \{\emptyset, \emptyset\}$. It represents the dynamic dependencies of the considered mixin module, and is therefore called a *dynamic graph*. It is used to detect ill-founded recursive value definitions. The graph $\rightarrow$ is an unlabeled graph over names. It represents the static dependencies of the considered mixin modules, and is therefore called a *static graph*. It is used to detect cyclic type definitions.

**Recursive definitions and type constraints** As usual, let rec binds variables to their values. It is required not to bind the same variable twice. For homogeneity, we consider these bindings as definitions. Moreover, names and fake dependencies are irrelevant in let rec so we do not write them. Any expression is allowed as a let rec definition, except that forward references must point to expressions of predictable shape, exactly as for MM in chapter 3. Expressions of predictable shape are defined by

$$e_\downarrow \in \text{Predictable} ::= \{o\} \mid \langle I; o \rangle \mid \text{let rec } b \text{ in } e_\downarrow.$$

The language allows to constrain the type of an expression $e$ by writing $(e : M)$. Notice that this operator is static, and is therefore only able to make some type declarations abstract, not to forget components.

**Operations on sequences** Lists and finite sets of definitions or declarations can be seen as finite maps from pairs of a label and a variable to different kinds of codomains. For instance, signatures are maps to types, outputs are maps to pairs of a set of fake dependencies and an expression. For each such map $f$, we denote by $DN(f)$ the set of names defined by $f$, $\{S \mid \exists s, (S, s) \in \text{dom}(f)\}$, and $DV(f)$ the set of variables defined by $f$, $\{s \mid \exists s, (S, s) \in \text{dom}(f)\}$.

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

**Dynamic semantics** The dynamic semantics is defined exactly as the semantics of MM in chapter 3, after removing all type indications.

### 5.2 Type system

The type system consists in four mutually dependent groups of relations: type well-formedness, matching and equivalence, and typing. Each of this group has a component for types, signatures, etc. They rely on the notion of environment $\Gamma$, referring to finite maps from variables to types. Environment bindings are defined as finite maps from pairs of a label and a variable to types. This way, a signature can be extracted from an environment, by removing all the anonymous bindings. Environment extension $+ \Gamma$ denotes the union of finite maps, without overriding. Therefore, $\Gamma + \Gamma'$ implies $DV(\Gamma) \downarrow DV(\Gamma')$. A signature can be seen as an environment by forgetting labels. This will be done implicitly in the following.
5.2.1 Well-formedness

The definition of well-formedness uses a new notion, the one of static dependency graph, which we introduce now.

Static dependency graph The static dependency graph \( \rightarrow_{\text{static}} \) of a set of declarations \( O \) with dynamic dependency graph \( \rightarrow \) is defined in figure 5.2. The arrow \( \rightarrow \) denotes any concrete dynamic graph: it is a graph over nodes, which are elements of \( \text{Vars} \cup \text{Names} \), labeled by degrees. Further, \( O \) is any set of declarations, not necessarily defining distinct names or variables. The definition of \( \rightarrow \) uses the function \( \text{Nodes} \), which associates to a pair \( (L, s) \) of a label and a variable either \( s \), if \( L = \ldots \) or \( L \) if \( L \) is a name. (By abuse of notation, we overload this function to act on declarations as well.) Roughly, this graph connects manifest type definitions referring to other type definitions in the same signature. It connects them by name when possible, and by variable otherwise. Formally, a declaration \( D_2 \) statically depends on another declaration \( D_1 \) if the type of \( D_2 \) contains a type declaration \( S \triangleright t : M \), such that the variable \( s \) defined by \( D_1 \) is free in \( M \).

Static graphs will often be required to be acyclic, which is written \( \vdash \rightarrow_{\text{static}} \). This condition avoids the difficulty of type-checking and type equivalence in the presence of equi-recursive, higher-order type constructors [29, 38]. Notice that the static graph is closed under dynamic dependency. This is necessary to rule out recursive types, as shown by the following example.

Consider \( e =_{\text{def}} \text{close}(T \triangleright t = y.U, X \triangleright x = \emptyset; U \triangleright u = t), Y \triangleright y = \text{close} x) \). The edges \( Y \rightarrow T \) and \( T \rightarrow X \) in the structure are obvious, but without the rules for prolongation with dynamic dependencies, there would not be any edge \( T \rightarrow Y \). The expression reduces to

\[
\text{let rec } T \triangleright t = y.U, \\
X \triangleright x = \emptyset; U \triangleright u = t, \\
Y \triangleright y = \text{let rec } U \triangleright u = t, \\
\quad \text{in } \{ U \triangleright u' = u \} \\
\text{in } \{ T \triangleright t' = t, X \triangleright x' = x, Y \triangleright y' = y \}
\]

which would have the recursive type \( \{ T \triangleright t : y.U, X \triangleright x : \{ \ldots \}, Y \triangleright y : \{ U \triangleright u : t \} \} \).

Well-formedness predicate Well-formedness of types, signatures, and declarations is defined as the least relation respecting the rules in figure 5.4, using figures 5.2 and 5.3. Notice that it depends on the typing relation.

A type variable \( t \) is a well-formed type, provided it is defined by the environment (rule \( \text{WF-Var} \)). If the path \( p \) has a module type exporting the type \( T \), then \( p.T \) is a well-formed type (rule \( \text{WF-Path} \)).

A module type is well-formed, provided the set of its declarations is well-formed in the environment extended with themselves (rule \( \text{WF-Module} \)), and provided that dependencies between type definitions (including the nested ones) are not cyclic. Formally, its static dependency graph \( \rightarrow_{\text{static}} \) is required to be acyclic. By rule \( \text{WF-Mixin} \), a mixin module type \( \langle I; O; \rightarrow \rightarrow \rangle \) is well-formed, provided the set \( I \) is well-formed in the environment \( \Gamma + I \), and the set of its output declarations is well-formed in the environment \( \Gamma + I + O \). Moreover, it is required that the union of the static graphs of \( I \) and \( O \) is acyclic, and that the dynamic graph \( \rightarrow \) is correct. A dynamic graph is said correct if its transitive closure is a partial ordering. The transitive closure of a dynamic graph \( \rightarrow \) is defined in figure 5.3, as the set of paths of \( \rightarrow \), labeled with the last edge of the path. This notion of graph correctness has been proven in chapter 4 to be a correct criterion for checking dependencies in mixin modules.

A well-formed signature is a signature containing only correct declarations. A declaration \( S \triangleright s : M \) is correct, provided \( M \) is (rule \( \text{WF-Manifest} \)), but the abstract type \( * \) is not a well-formed type by itself. Rule \( \text{WF-Abstract} \) allows a declaration to use the abstract type, but only for type declarations. Notice that the abstract type \( * \) is not the type of any value definition. In fact, the abstract type can be seen as a syntactic artefact to include abstract and manifest type declarations into a single syntactic class. The notion of well-formed environments is derived from the one for signatures.
**Static free variables**

\[
\begin{align*}
SFV(\star) &= \emptyset \\
SFV(t) &= \emptyset \\
SFV(x, p, T) &= \emptyset \\
SFV(\{O\}) &= \bigcup_{D \in O} SFV(D) \setminus DV(O) \\
SFV((I; O; \rightarrow; \rightarrow)) &= \bigcup_{D \in I} SFV(D) \setminus DV(I) \cup \\
&\quad \bigcup_{D \in O} SFV(D) \setminus DV(I, O)
\end{align*}
\]

**Static dependency graph**

\[
\begin{align*}
\text{Node}(D_{i_1}) \rightarrow (\{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}, \rightarrow) &\rightarrow \text{Node}(D_{i_2}) \\
\text{Node}(D_{i_1}) \rightarrow (\{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}, \rightarrow) &\rightarrow \text{Node}(D_{i_3}) \\
\text{Node}(D_{i_1}) \rightarrow (\{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}, \rightarrow) &\rightarrow \text{Node}(D_{i_5}) \\
s \in SFV(D_{i_2}) &\rightarrow s = DV(D_{i_1}) \\
\text{Node}(D_{i_1}) \rightarrow (\{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}, \rightarrow) &\rightarrow \text{Node}(D_{i_2})
\end{align*}
\]

**Figure 5.2:** Static dependencies in a signature

\[
\begin{align*}
X \xrightarrow{\lambda_{i_1}^+} Z &\quad Z \xrightarrow{\lambda_{i_2}^+} Y \\
X \xrightarrow{\lambda_{i_2}^+} Y &\quad X \xrightarrow{\lambda_1^+} Y
\end{align*}
\]

**Figure 5.3:** Transitive closure of $\rightarrow$
5.2.2 Typing

The typing rules are in figure 5.9, and they use figures 5.5 to 5.8.

Rule TT-VAR gives a variable the type M proposed by the environment, strengthened as explained in figure 5.5. Type strengthening [51], sometimes also called selfification [40], consists, when using a module type bound to a variable x in the environment, in keeping track of where its abstract types come from. The way it is done is by replacing abstract types with manifest types indicating that they come from x. For instance, if x is bound in the environment to the module type \( \{ T \triangleright t : * \} \), then x has type \( \{ T \triangleright t : x.T \} \). Because of nested modules, the operation more generally consists in prefixing the abstract type names with x, followed by the access path inside the module.

Rule TT-ACCESS explains how a computational component is accessed in a module. Assume x has type \( \{ T \triangleright u : M, Y \triangleright y : u \} \). The type of x.Y cannot simply be u, because the type variable u would escape its scope. The system has to find a type equivalent to u. It is done by accessing the path to u, i.e. giving x.Y the type x.T. Formally, this is done by an operation called extraction, and defined as the substitution

\[ [O \rightarrow p.O] = \{ s \mapsto p.S \mid (S, s) \in \text{dom}(O) \} \]
Dynamic free variables

\[ \text{DFV}(x) = \{ x \} \]
\[ \text{DFV}(x, p, X) = \{ x \} \]
\[ \text{DFV}(\{ o \}) = \bigcup_{d \in o} \text{DFV}(d) \setminus \text{DV}(o) \]
\[ \text{DFV}(\text{let rec } o \text{ in } e) = (\bigcup_{d \in e} \text{DFV}(d) \cup \text{DFV}(e)) \setminus \text{DV}(o) \]
\[ \text{DFV}(\langle I; o \rangle) = (\bigcup_{d \in e} \text{DFV}(d)) \setminus \text{DV}(o) \setminus \text{DV}(I) \]
\[ \text{DFV}(e_1 + e_2) = \text{DFV}(e_1) \cup \text{DFV}(e_2) \]
\[ \text{DFV}(\text{close } e) = \text{DFV}(e) \]
\[ \text{DFV}(\langle e : M \rangle) = \text{DFV}(e) \]

Dynamic dependency graph

\[
\chi = \text{Degree}(x', e) \quad (L', x') \in \text{dom}(I) \cup \text{dom}(o) \quad (L[x^*] \triangleright x = e) \ni o
\]
\[
\text{Node}(L', x') \xrightarrow{\chi(I; o)} \text{Node}(L, x)
\]
\[
\text{Node}(L_i, x_i) \text{ dom}(I) \cup \text{dom}(o) \quad (L[x_1 \ldots x_n] \triangleright x = e) \ni o
\]
\[
\text{Node}(L_i, x_i) \xrightarrow{\chi(I; o)} \text{Node}(L, x)
\]

Figure 5.6: Dynamic dependencies in a structure

Rule TT-STRUCT allows to type structures \( \langle I; o \rangle \). A type has to be guessed for each definition, and these types are grouped together in an environment \( \Gamma_o \), where the names have been kept from \( o \). This environment is checked well-formed. The type of the structure is obtained by forgetting the anonymous declarations in \( \Gamma_o \), yielding a signature \( O_o \). But \( O_o \) still might depend on the anonymous definitions. The system thus has to find a super signature \( O \) of \( O_o \), that prevents variables from escaping their scopes. Intuitively, eliminating a reference to a local type definition is done as follows. For a reference to a type abbreviation, it consists in expanding it recursively (acyclicity guarantees termination). For a reference to an abstract type, if an exported abbreviation to it has already been made, then refer to this abbreviation, and otherwise return the abstract type.

The involved signatures and environment are checked well-formed; the static dependency graph \( \rightarrow_{(I; o)} \) is checked acyclic; and the concrete dependency graph of \( \langle I; o \rangle \), denoted by \( \rightarrow_{(I; o)} \), is checked correct and lifted, as explained below.

The concrete dependency graph of a structure is defined in figure 5.6. It is a graph over nodes, which are elements of \textit{Vars} \cup \textit{Names}. Edges may be defined in two ways. First, a definition \( d = (L \triangleright x[x_1 \ldots x_n] = e) \) specifies a fake dependency on each \( x_i \), so for each \( i \), if \( (L_i, x_i) \in \text{dom}(o) \cup \text{dom}(I) \), then there is an edge \( \text{Node}(L_i, x_i) \xrightarrow{\otimes} \text{Node}(L, x) \). Second, if the body \( e \) of a definition \( d = (L \triangleright x[x_1 \ldots x_n] = e) \) dynamically depends on a variable \( x' \), such that \( (L', x') \in \text{dom}(I) \cup \text{dom}(o) \), then there is an edge from \( \text{Node}(L', x') \) to \( \text{Node}(L, x) \). The notion of dynamic dependence is defined in figure 5.6, and roughly corresponds to forgetting type constraints. The degree of the edge is \textit{Degree}(x, e), where the \textit{Degree} function is defined for \( x \in \text{DFV}(e) \) by

\[
\text{Degree}(x, \langle I; o \rangle) = \emptyset
\]
\[
\text{Degree}(x, \{ o \}) = \emptyset
\]
\[
\text{Degree}(x, e) = \emptyset \text{ otherwise.}
\]

When this concrete dependency graph has been checked correct, in the sense that its transitive closure restricted to strict edges is a partial ordering, it can be lifted to an abstract dependency graph. This operation consists in prolonging edges to local definitions until they reach an exported one, and then forgetting the edges involving local definitions. It is described in figure 5.7.
Lift

Transitive closure through local components

\[
\begin{align*}
N_1 \xrightarrow{x_1} x & \xrightarrow{x_2}^\triangledown N_2 \\
N_1 & \xrightarrow{x_1 \land x_2}^\triangledown N_2
\end{align*}
\]

\[
\text{Lift: } \rightarrow = \rightarrow^\triangledown | \text{Names } \times \text{Names}
\]

Figure 5.7: Lifting concrete dependency graphs

\[(I_1, O_1) \Join (I_2, O_2) \text{ means } \left\{ \begin{array}{l}
I_1 \sqsubseteq_{O_2} I_2 \text{ and } \\
I_2 \sqsubseteq_{O_1} I_1.
\end{array} \right.\]

\[I_1 \sqsubseteq O I_2 \text{ means that for all } (L, s) \in \text{dom}(I_1),
\]
\[x \in \text{FV}(I_2, O) \cup \text{DV}(O, I) \Rightarrow (L, s) \in \text{dom}(I_2) \text{ and } L \in \text{Names}.\]

Figure 5.8: Compatibility

Rule TT-Module types basic modules \(\{o_i\}\), as if it were a mixin with no input declaration, except that given the restricted form of definitions allowed, it is simpler.

The rule TT-LetRec is as the TT-Struct for mixin modules without input declarations, for the binding part at least. The final expression is then typed in the context extended with the most precise signature available for the bindings, and the obtained type must not allow variables to escape their scopes.

Rule TT-Composition types the composition of two mixin modules of type \(\langle I_i; O_i; \rightarrow_i; \gamma_i \rangle\), for \(i = 1, 2\). The two mixin module types are first checked compatible, as defined in figure 5.8. Roughly, it ensures that variables are not captured during composition. Then, the unions of the two static and dynamic dependency graphs must be correct. They will be the dependency graphs of the final type. Its output signature is the disjoint union of the two output signatures \(O_1\) and \(O_2\), in the sense that they must not define the same name twice. The input signature of the final type is a new signature \(I\), which must be a sub signature of both \(I_1\) and \(I_2\). This way, the requirements made on inputs in the composition are stronger than in each argument, thus preserving type safety. The signature \(I\) could introduce edges in the static dependency graph, so the final graph is checked acyclic.

Rule TT-Constraint defines type constraints. For typing \((e : M)\), assuming \(e\) has type \(M'\), it is checked that \(M'\) is a subtype of \(M\), and if so, the type of \((e : M)\) is \(M\).

Finally, the TT-Close rule types mixin module instantiation. A mixin module of type \(\langle I; O; \rightarrow; \gamma \rangle\) is instantiated as follows. Semantically, the variables defined by \(O\) must replace the input variables of \(I\). This is done by the substitution \(\sigma\), and we obtain two signatures \(I'\) and \(O'\). It is then checked that in the environment extended by \(O'\), the signature \(O'\) matches the signature \(I'\).

5.2.3 Subtyping

It is easy to see that forgetting some output fields in mixin modules would be dangerous: the well-known problems with width subtyping of extensible records (see e.g. [42]) can be encoded with mixin modules. The first of these problems happens with composition +, putting two components with the same name in conflict. For example, the expression \(\langle \emptyset; X \triangleright x = \{\} \rangle + (\emptyset; X \triangleright x = \{\} : \emptyset; \emptyset; \emptyset)\) is stuck. The second problem arises with the overriding operator of section [?], when one field is overridden with a field with the same name, but a different type, as in
\[ \Gamma(x) = M \quad \frac{\Gamma \vdash x : M | x}{\text{(TT-VAR)}} \quad \frac{\Gamma \vdash p : \{O\}}{\text{(TT-ACCESS)}} \]
\[ \Gamma + I \vdash I \quad \Gamma + I + O \vdash O \quad \frac{\frac{\Gamma + I + O \vdash \rightarrow_{\langle I, o \rangle}}{\text{(TT-STRUCT)}}}{\frac{\Gamma + I + \Gamma_o \vdash \langle I, o \rangle \vdash \rightarrow_{\langle I, o \rangle}}{\text{(TT-STRUCT)}}} \]
\[ \frac{\Gamma + I + \Gamma_o \vdash o : \Gamma_o}{\frac{\Gamma + I + \Gamma_o \vdash \rightarrow_{\langle I, o \rangle}}{\text{(TT-STRUCT)}}} \quad \frac{\Gamma + \Gamma_o \vdash e : M'}{\frac{\Gamma + \Gamma_o \vdash e : M'}{\text{(TT-LETREC)}}} \quad \frac{\frac{\frac{\Gamma + \Gamma_o \vdash M}{\Gamma + \Gamma_o \vdash o : \Gamma_o}}{\text{(TT-STRUCT)}}}{\frac{\text{let } o \text{ in } e : M}{\text{(TT-LETREC)}}} \]
\[ \frac{\Gamma \vdash e_1 : \langle I_1 ; O_2 ; \rightarrow_1 ; \rightarrow_1 \rangle}{\frac{\Gamma \vdash e_2 : \langle I_2 ; O_2 ; \rightarrow_2 ; \rightarrow_2 \rangle}{\text{(TT-COMPOSITION)}}} \quad \frac{\Gamma + I \vdash I \leq I_1}{\frac{\Gamma + I \vdash I \leq I_2}{\text{(TT-COMPOSITION)}}} \quad \frac{\frac{\frac{\Gamma + I \vdash e_1 + e_2 : \langle I ; O_1 + O_2 ; \rightarrow_1 + \rightarrow_2 ; \rightarrow_1 + \rightarrow_2 \rangle}{\text{(TT-COMPOSITION)}}}{\frac{\Gamma + M \vdash M'}{\text{(TT-CONSTRAINT)}}} \frac{\Gamma + M' \leq M}{\text{(TT-CONSTRAINT)}}}{\frac{\Gamma \vdash e : M}{\text{(TT-CONSTRAINT)}}} \]
\[ \frac{\Gamma \vdash e : \langle I ; O ; \rightarrow ; \rightarrow \rangle}{\frac{\sigma = \{ s \mapsto s' \mid (S, s) \in \text{dom}(I), (S, s') \in \text{dom}(O) \}}{\text{(TT-CLOSE)}}} \quad \frac{\Gamma = I(\sigma)}{\text{(TT-CLOSE)}} \quad \frac{O' = O(\sigma)}{\frac{\Gamma + O' \vdash \{ O' \} \leq \{ I' \}}{\text{(TT-CLOSE)}}} \]
\[ \forall 1 \leq i \leq n, \Gamma \vdash d_i : D_i \quad \frac{\Gamma \vdash (d_1 \ldots d_n) : (D_1 \ldots D_n)}{\text{(TT-OUTPUT)}} \quad \frac{\Gamma \vdash e : M}{\frac{\Gamma \vdash (X \triangleright x'[y']) = e : (X \triangleright x : M)}{\text{(TT-EXPR)}}} \]
\[ \frac{\Gamma \vdash M}{\frac{\Gamma \vdash (T \triangleright t : M) : (T \triangleright t : M)}{\text{(TT-TYPE)}}} \]

Figure 5.9: Typing rules
### Types

$$\begin{align*}
\Gamma &\vdash M_1 \cong M_2 \quad \text{(ST-EQUIV)} \\
\Gamma &\vdash M_1 \leq M_2
\end{align*}$$

$$\begin{align*}
\Gamma &\vdash \{O_1 \vdash O_1' \leq O_2\} \\
\Gamma &\vdash \{O_1 \vdash O_1' \leq \{O_2\}\}
\end{align*}$$

$$\begin{align*}
\Gamma &\vdash I_2 + I_2 + O_1 \vdash O_2 \\
\Gamma &\vdash \{I_1 ; O_1 \vdash \rightarrow \rightarrow \rightarrow_1 \leq \rightarrow \rightarrow_2\} \leq \{I_2 + I_2'; O_2 ; \rightarrow \rightarrow_2\}
\end{align*}$$

### Signatures

$$\begin{align*}
\forall i \leq n, \Gamma &\vdash D_i \leq D'_i \quad \text{(ST-Sig)}
\end{align*}$$

### Declarations

$$\begin{align*}
\Gamma &\vdash M_1 \leq M_2 \\
\Gamma &\vdash (X \triangleright x : M_1) \leq (X \triangleright x : M_2) \quad \text{(ST-VAL)}
\end{align*}$$

$$\begin{align*}
\Gamma &\vdash \{T \triangleright t : M\} \leq \{T \triangleright t : \ast\} \quad \text{(ST-CON-ABS)}
\end{align*}$$

$$\begin{align*}
\Gamma &\vdash t \cong M \\
\Gamma &\vdash \{T \triangleright t : \ast\} \leq \{T \triangleright t : \ast\} \quad \text{(ST-ABS-ABS)}
\end{align*}$$

$$\begin{align*}
\Gamma &\vdash M_1 \cong M_2 \\
\Gamma &\vdash \{T \triangleright t : M_1\} \leq \{T \triangleright t : M_2\} \quad \text{(ST-CON-CON)}
\end{align*}$$

### Figure 5.10: Subtyping and signature matching

$$\begin{align*}
\{\emptyset; X \triangleright x = \{Y \triangleright y = x'\},
\_	riangleright z = x \triangleright Y
\} : \{\emptyset; \emptyset; \emptyset\}
\end{align*}$$

which reduces to the ill-typed $$\{\emptyset; X \triangleright x = \{Y \triangleright y = x'\}, \_	riangleright z = x \triangleright Y\}$$

Subtyping is the least transitive relation respecting the rules in figure 5.10. For output components of mixin modules, this relation corresponds to depth subtyping: it allows some type declarations to be made abstract, and some value declarations to be made less precise, but no declaration can be forgotten. In input signatures, it is possible to add some deferred components. This appears especially in rule ST-Sig, where declarations must be in a one-to-one correspondence. For modules, however, it is allowed to forget some output components.

Subtyping is reflexive modulo type equivalence by rule ST-EQUIV. Rule ST-MODULE allowed to change its signature for a less precise one. By rule ST-MIXIN, a mixin module is more precise if it input signature is less precise, i.e. puts less requirements on inputs, and its output signature is more precise, i.e. it provides more capabilities. Also, by definition, the notion of graph subtyping allows to add edges in the graph, and to change of labels into the ones.

Signature comparison, is made declaration by declaration. A value declaration may be replaced with a value declaration of less precise type (rule ST-VAL). Any type declaration can be made abstract (rules ST-CON-ABS and ST-ABS-ABS) A manifest type declaration can be replaced with an equivalent manifest type declaration. An abstract type declaration $$T \triangleright t : \ast$$ can be replaced with a manifest type declaration $$T \triangleright t : M$$ provided the type $$t$$ is provably equivalent to $$M$$. This can happen for example, when comparing two equivalent but differently ordered module types, such as $$\{T \triangleright t : \ast, U \triangleright u : t\}$$ and $$\{U \triangleright u : \ast, T \triangleright t : u\}$$. This leads to comparing the declarations $$T \triangleright t : \ast$$ and $$T \triangleright t : u$$ in the environment $$T \triangleright t : \ast, U \triangleright u : t$$, where $$u$$ is provably equal to $$t$$.

98
### Types

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TE-REFL</strong></td>
<td>$\Gamma \vdash M \iff M$</td>
</tr>
<tr>
<td><strong>TE-VAR</strong></td>
<td>$\Gamma \vdash t \iff \Gamma(t)$</td>
</tr>
<tr>
<td><strong>TE-ACC</strong></td>
<td>$\Gamma \vdash p : {O} \quad O(T) \neq * \quad \Gamma \vdash p.T \equiv O(T)[O \mapsto p.O]$</td>
</tr>
<tr>
<td><strong>TE-MODULE</strong></td>
<td>$\Gamma \vdash \text{O}_1 \equiv \text{O}_2$</td>
</tr>
<tr>
<td><strong>TE-MIXIN</strong></td>
<td>$\Gamma \vdash \text{I}_1 ; \text{O}_1 ; \rightarrow_1 ; \rightarrow_1 \equiv \langle \text{I}_2 ; \text{O}_2 ; \rightarrow_2 ; \rightarrow_2 \rangle$</td>
</tr>
</tbody>
</table>

### Signatures

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TE-SIG</strong></td>
<td>$\forall 1 \leq i \leq n, \Gamma \vdash D_i \equiv D'_i \quad \Gamma \vdash {D_1 \ldots D_n} \equiv {D'_1 \ldots D'_n}$</td>
</tr>
</tbody>
</table>

### Declarations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TE-COMP</strong></td>
<td>$\Gamma \vdash M_1 \equiv M_2 \quad \Gamma \vdash (S \triangleright s : M_1) \equiv (S \triangleright s : M_2)$</td>
</tr>
</tbody>
</table>

---

**Figure 5.11: Type equivalence**

### 5.2.4 Type equivalence

Type equivalence is the least symmetric and transitive relation respecting the rules in figure 5.11. It is not reflexive, because the abstract type is not equal to itself (fortunately for type soundness), but calling *determinate* types the types different from it, type equivalence is reflexive on determinate types.

A type variable is equivalent to the type it has been assigned by the environment (rule TE-VAR). If a path $p$ has a module type exporting a type declaration $T \triangleright t : M$, then by the typing rule TT-VAR, this module type has been strengthened, so $M$ is determinate, and $p.T$ is equivalent to the extraction of $M$.

Then, module and mixin module types are defined straightforwardly through the notion of signature equivalence, which checks the equivalence of the types associated to declarations, in a one-to-one correspondence.

### 5.2.5 Directions for a proof of soundness

**The problem**  In MML, as in any type system with type abstraction, type soundness is hard to prove because type abstraction invalidates type preservation. The problem is easy to see. Assume a module `A` has been defined, in an OCaml-like syntax, by

```ocaml
module A = (struct
type t = int
  let x = 1
end : sig
type t
  val x : t
end)
```
(Here, the construction \texttt{(module : module-type)} denotes the coercion of a module to a module type.)

The module \texttt{A} is bound in the typing environment with the type

\begin{verbatim}
 sig
type t
val x : t
end
\end{verbatim}

Now, if further in the program we use \texttt{A.x}, then its type is simply \texttt{A.t}, not \texttt{int}. Indeed, in the type of \texttt{A}, no definition is provided for \texttt{t}. Until now, no difficulty arose, but if we try to evaluate our program, then \texttt{A.x} evaluates to 1, which is of type \texttt{int}, but not of type \texttt{A.t}: type preservation does not hold.

\textbf{Lillibridge’s solution}  \textnormal{Lillibridge [56] defines a kernel module system called the \textit{translucent sums} formalism, apparently close to the manifest types formalism, but which enjoys the type preservation property. We illustrate the subtle differences leading to this result, and their consequences.}

Let \texttt{m} \texttt{=} \texttt{def} \texttt{struct \ and \ S} \texttt{=} \texttt{def} \texttt{sig}
\begin{verbatim}

type t = int
let x = 1
val x : t
end
\end{verbatim}

The counter-example program showing that type preservation does not hold in the manifest types approach is \texttt{module A = \{m : S\}, let res = A.x}. Recall that in the manifest types approach, this expression is well-typed, and \texttt{res} has type \texttt{A.t}. The problem is that during evaluation its type changes. In the translucent sums approach, there is no primitive let binding, so one has to encode the program as a functor application, taking advantage of the fact that the principal type of \texttt{(m : S)} is known to be \texttt{S: (functor (A : S) = struct let res = A.x end) m}.

In Lillibridge’s system, this expression is well-typed. Indeed, \texttt{m} has type \texttt{sig type t = int val x : int end}. Moreover, the functor has type \texttt{functor (A : sig type t val x : t end) \rightarrow sig val res : A.t end} as a principal type. This type is a subtype of \texttt{functor (A : sig type t = int val x : int end) \rightarrow sig val res : int end}. Therefore, by subsumption, the functor can be given this type. As it is a non-dependent functor type, the whole program has type \texttt{int}. Therefore, when selection is performed, this type is preserved.

However, one could object that we cheated a bit here, by forgetting that \texttt{m} was initially coerced to \texttt{S}. And indeed, if we replace \texttt{m} with \texttt{m : S} in our encoding, we obtain an ill-typed expression. Indeed, the principal type of the argument to the functor, \texttt{m : S}, is \texttt{S}, which is not transparent. The consequence is that the functor cannot be specialized, as above, to a non-dependent type, and therefore the program is ill-typed.

\textbf{Related approaches}  \textnormal{In [31], Duggan and Sourelis prove the soundness of their calculus of mixin modules by showing the soundness of the calculus without type abstraction by explicit coercion, and remarking that each well-typed term in the presence of abstraction is well-typed without type abstraction. The restricted calculus strongly resembles Lillibridge’s kernel system. The only type abstraction lies in functor abstraction. Lillibridge’s system retains explicit coercion, but its use is limited by the type system. Courant [23, 24] adds type equalities to the type theory of its module calculus in order to retain type preservation.}

100
**Syntactic type abstraction** A drawback of these approaches is that while retaining the important property of type preservation, it is difficult to prove that abstraction is preserved during evaluation. For instance, once the argument module is passed to a functor, the type system forgets that it possibly had some abstract types. Such abstraction properties as representation independence have been proven by Mitchell [59], from a denotational semantics standpoint, but they are reported by Grossman et al. [39] to extend with difficulty to new language features. Instead, Grossman et al. propose a new, syntactic technique for proving abstraction properties of systems, which scales well to new language features. It is based on embeddings for exporting abstract values outside of the scope of abstraction. The authors notice as an interesting future work that this technique might apply to module systems.

### 5.2.6 Undecidability, principal types, syntactic types

**Conjecture of undecidability** We conjecture that the typing MML is undecidable, based on Lillibrige’s result that typing the OCaml module system is [56].

**Conjecture 1 (Undecidability)** Signature matching is undecidable in MML.

The following example in OCaml syntax gives an idea why the intuitive algorithm fails for modules. It is easy to encode this example with functors. We refer to Lillibrige’s thesis for more details.

```ocaml
module type I = sig
  module type A
  module F : functor(X : sig
    module type A = A
    module F : functor(X : A) -> sig end
  end) -> sig end
end ;;

module type J = sig
  module type A = I
  module F : functor(X : I) -> sig end
end ;;

module Loop(X : J) = (X : I) ;;
```

The intuitive algorithm fails, because for matching J against I, it puts the components of J in the environment, thus making the module type component A in I equal to I itself. Thus, when contravariantly matching the arguments of the functor components F of each module type, it in fact matches J against I, once again.

**Principal types** A type system has principal types if given an environment and an expression, there exists a minimal type such that the expression has this type in the given environment. We do not know whether MML has principal types.

**Syntactic types** For separate compilation, it is desirable for the programmer to be able to express any signature of the language, syntactically. Indeed, it allows to put as much information as needed in interfaces. Several known module systems do not have syntactic signatures, e.g. the ones of Russo [65], Dreyer et al. [28], or the one of OCaml. For example, in OCaml, external names are not distinguished from internal variables. It is thus impossible to express the type

```ocaml
{ type T > t : *,
  val X > x : {type T > t' = t} }
```
without changing the names of some components.

A concrete system implementing MML would probably make the same choice of identifying external names and internal variables, and would thus lack syntactic types.

5.3 Polymorphism and datatypes

The formalism already encodes explicit polymorphism [37] and is easily extended with datatypes in the style of ML [58].

5.3.1 Polymorphism

As in [40], polymorphism is encoded by our formalism, although it is only explicit polymorphism. We use the following syntactic sugar conventions, where \( ARG, RES \in MNames \), \( arg, res \in MVars \), \( TARG \in TNames \), and \( targ \in TVars \). The variables \( arg \), \( targ \) and \( res \) are not allowed to occur free anywhere, and the names \( ARG \), \( TARG \) and \( RES \) are reserved.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ( \lambda x : M. e )</td>
<td>( (ARG \triangleright x : M; RES \triangleright res = e) )</td>
</tr>
<tr>
<td>Application ( e_1 e_2 )</td>
<td>( \langle ARG \triangleright arg = e_1; ARG \triangleright arg = \emptyset; res \rangle )</td>
</tr>
<tr>
<td>Function type ( M_1 \rightarrow M_2 )</td>
<td>( \langle ARG \triangleright arg : M_1; RES \triangleright res : M_2 \rangle )</td>
</tr>
<tr>
<td>Type function ( \forall t. M )</td>
<td>( \langle TARG \triangleright t : \star; RES \triangleright res = e \rangle )</td>
</tr>
<tr>
<td>Type application ( e[M] )</td>
<td>( \langle arg = \emptyset; TARG \triangleright targ = M \rangle )</td>
</tr>
<tr>
<td>Type function type ( \forall t. M )</td>
<td>( \langle TARG \triangleright t : \star; RES \triangleright res : M; \emptyset; {TARG \triangleright res } \rangle )</td>
</tr>
</tbody>
</table>

5.3.2 Datatypes

It is not too difficult to add ML-like datatypes to MML. ML datatypes are user-defined abstract types, accompanied with a finite list of constructors, which allow to build values of that type.

Background In [26], Crary et al. study the interpretation of Standard ML datatypes in type theory. They propose two possible interpretations, the opaque and the transparent interpretations.

Intuitively, the opaque interpretation is the one of Standard ML: a datatype is interpreted as a new type, and values of that type can only be created by application of the associated constructors. For example, the OCaml signature

\[
S_1 \quad =_{\text{def}} \quad \text{sig}
\begin{align*}
\text{type} & \quad u = A \ \text{of} \ u \times u \ \mid \ B \ \text{of} \ \text{int} \\
\text{type} & \quad t = u \times u
\end{align*}
\text{end}
\]

is interpreted as

\[
S_1^{\text{opaque}} \quad =_{\text{def}} \quad \text{sig}
\begin{align*}
\text{type} & \quad u \\
\text{val} & \quad _{\text{in}} : (u \times u + \text{int}) \to u \\
\text{val} & \quad _{\text{out}} : u \to (u \times u + \text{int})
\end{align*}
\text{end}
\]
This interpretation is used in [43], which gives a formal interpretation of Standard ML into type
theory. Nevertheless, Crary et al. reject it because each datatype construction or pattern-matching
corresponds to the run-time cost a function call. Instead, they propose to use the transparent
interpretation, in which a datatype is rather interpreted as a recursive sum type. The signature
$S_1$ is interpreted as
\[
S_1^{\text{transparent}} = \text{def} \quad \text{sig}
\]
\[
\begin{align*}
type \ u &= \mu \ u . (u \ast u) + \text{int} \\
type \ t &= u \ast u
\end{align*}
\]
end

Notice that there is no need for introducing special constructors, as $u_{\text{in}}$ and $u_{\text{out}}$ in the opaque
interpretation, since one can rely on the sum type injections to produce values of type $u$. Fur-
more, the recursive type constructor $\mu$ is difficult to deal with. In their papers on recursive
modules [27, 29], Harper et al. study two possible type theoretic constructions implementing $\mu$,
distinguishing \textit{equi-recursive} types from \textit{iso-recursive} types.

In the equi-recursive approach, the type $\tau = \mu \ u . (u \ast u) + \text{int}$ above is equivalent to its
unrolling $(\tau \ast \tau) + \text{int}$. To construct a value of type $\tau$, construct a value of type $\text{int}$ or $\text{int}
\ast \tau$, and just inject it into the sum type, thanks to the left and right injections $\text{inj}$ and $\text{inj}$,
respectively. For example, $e = \text{def} \ inj \ 1$ has type $\tau$. Such expressions are decomposed by the
projection operations of sum types, $\text{proj}$ and $\text{proj}$, so one can recover the integer from $e$ by $\text{proj} \ e$.

In the iso-recursive approach, $\tau$ is only isomorphic to its unrolling $(\tau \ast \tau) + \text{int}$. Concretely,
this means that given some term $e'$ of type $(\tau \ast \tau) + \text{int}$, there is a rolling operation roll that
coerces $e'$ to $\tau$: roll $e'$ is of type $\tau$. Conversely, to use a value of type $\tau$, one has to apply the unroll
operation first, which coerces it to $(\tau \ast \tau) + \text{int}$. For instance, to construct a value of type $\tau$, one
wants $e = \text{def} \ \text{roll}(\text{inj} \ 1)$, and its first element is accessed through $(\text{unroll}(\text{proj} \ e))$.

The tension lies between the expressivity of the equi-recursive approach and the fact that it makes
type equivalence possibly undecidable. Conversely, the iso-recursive approach is a bit less flexible,
but retains decidability. In [26], Crary et al. choose the iso-recursive approach. However, the
transparent, iso-recursive interpretation of datatypes is not compatible with Standard ML, as
shown by the following example. In Standard ML, the signature $S_1$ is a subsignature of $S_2$, defined
as follows:
\[
S_2 = \text{def} \quad \text{sig}
\]
\[
\begin{align*}
type \ t \\
type \ u &= A \ of \ t \ | \ B \ of \ \text{int}
\end{align*}
\]
end

In the transparent, iso-recursive interpretation, it is not the case. Indeed, in order to prove it,
one has to prove that, assuming $u = \tau$ and $t = u \ast u$, the type $\tau$ is equivalent to $t + \text{int}$. It is
possible, by replacing $u$ with its value, to prove that $t$ is equivalent to $\mu \ u . (\tau \ast \tau) + \text{int}$, but
this type is not equivalent to $\tau$. In order to solve the problem, Crary et al. enrich the type system
with \textit{Shao’s equation}:
\[
\mu \alpha . \tau = \mu \alpha . (\tau[\alpha \mapsto (\mu \alpha . \tau)]) \quad (\text{SHAO})
\]

This allows to recover Standard ML datatypes.

We choose yet another approach, closer to \textit{inductive types} than to recursive types [77]. A datatype
definition is initially not considered equal to any type. It is rather defined by a list of \textit{constructors},
as the smallest type such that the only way to construct values of this type is to apply one of the
constructors. This method in fact closely corresponds to ML datatypes, and was added by Werner
[77] to the \textit{calculus of constructions} [22] for making the extraction of programs from proofs more
efficient. Thus, it can be considered as a perfectly type theoretical construction.
Formalization  Figure 5.12 extends MML with datatypes (with an approach inspired by [31, 66]). Assume given an infinite, denumerable set of constructor names \( C \in \text{ConNames} \). The notions it defines are mutually recursive with the ones of figure 5.14. Type paths \( pt \) are either type variables or type names prefixed by a module path. Expressions are extended with constructor applications \( C^{pl}[e_1 \ldots e_n] \), consisting in a constructor name, applied to a list of expressions, and annotated by the type path the constructor comes from. The list of arguments must match the arity of the constructor exactly, as will be enforced by the type system. Such an application is valid only when the constructor has been previously introduced by a new form of definition, called datatype definition, which has the shape \( T \triangleright t = \Phi \). Expressions are also extended with a family of operators for pattern-matching. The family is denoted by \( \text{match}^p \). It is indexed by a type path \( pt \), and a datatype \( \Phi \). Indexing the pattern-matching operators with datatypes allows to easily define their dynamic semantics. Indexing them over type paths is useful during typing, for checking that the datatype has been declared as indicated by \( \Phi \). A datatype \( \Phi = \phi_1 \ldots \phi_n \) is a list of constructor definitions, syntactically required not to bind the same constructor name twice, and a constructor definition \( \Phi = C[M_1 \ldots M_n] \) is a pair of the name of the new constructor, plus the list of its argument types. When the constructor is applied, its arguments are required by the type system to have these types. Notice that both the order of the constructors in a datatype definition and the order of the types in a constructor definition matter. At the level of types, declarations are enriched to take datatype definitions into account. A datatype definition \( T \triangleright t = \Phi \) corresponds to two declarations: one defines the new abstract type \( T \triangleright t = \ast \), while the other specifies its constructors \( t \prec \Phi \).

The set of expressions of syntactically predictable shape is extended with constructor applications \( C^{pl}[e_1 \ldots e_n] \), as shown in figure 5.12. A constructor declaration has no static free variables, since it does not define any type. It is well formed, provided the types it mentions are and it does not define the same constructor twice. The well-formedness condition on signatures now checks that only one unfolding \( t \prec \Phi \) is defined for each \( t \). Moreover, such \( t \) must be defined in the same signature, either as abstract types, or as types that unfold (see below) to an equivalent (see below) datatype. This flexibility is necessary, since by type strengthening abstract types are soon replaced with type paths. Type strengthening has no effect on an unfolding specification, rather on the associated type definition. Finally, the degree of a variable in a constructor application can be \( \emptyset \), if it is \( \emptyset \) in all arguments.

Dynamic semantics  Extending the dynamic semantics to handle constructors and pattern-matching is described by figure 5.13. First, values are extended with constructed values, that is, a constructor applied to values, and with partial matchings. The matching operator \( \text{match}^p \) expects the argument to the matching, plus \(|\Phi|\) functions for dealing with each of the constructors defined by \( \Phi \). When the final argument has not been provided, and the first arguments are evaluated, the expression is called a partial matching, and considered a value. As soon as the final argument is given, rule MATCH performs the matching. If the matching operator is \( \text{match}^p \), and the first argument to the matching is \( C^{pl}[v_1 \ldots v_n] \), according to the index of \( C_i \) in \( \Phi \), the rule applies one of the matching functions \( v_1 \ldots v_n \) to the arguments \( v'_1 \ldots v'_n \).

Static semantics  As shown by figure 5.14, the static semantics of MML is extended to account for datatypes. A new judgment, type unfolding \( t \prec \Phi \), is introduced, for retrieving the datatype corresponding to a type path. If it is simply a type variable \( t \), then an unfolding declaration \( t \prec \Phi \) must be in the environment. Otherwise, it is a type path \( pT \), then the datatype has to be extracted from the type of \( p \).

Typing constructor application \( C^{pl}[e_1 \ldots e_n] \) (rule TT-CONAPP) consists in unfolding the type path annotation \( pt \), to retrieve the corresponding datatype \( \Phi \), and check that the arguments match the types expected by \( \Phi \). Typing a matching operator \( \text{match}^p \) is a bit more complicated. There are two main checks to do: first, the \( pt \) annotation must unfold to a datatype \( \Phi \), and second the \( \Phi' \) annotation must be equivalent to that \( \Phi \). Then, the type of \( \text{match}^p \) is a polymorphic type \( \forall t.M \), where \( M \) is a function expecting the first argument of type \( pt \), plus the \(|\Phi|\) matching
### Syntax

**Type path:**
\[ pt ::= t \mid p.T \]

**Expression:**
\[ e ::= \ldots \mid C^{pt}[e_1 \ldots e_n] \mid \text{match}^pt \]

**Definition:**
\[ d ::= \ldots \mid T \triangleright t = \Phi \]

**Declaration:**
\[ D ::= \ldots \mid t \prec \Phi \]

**Datatype definition:**
\[ \Phi ::= \phi_1 \ldots \phi_n \]
\[ \phi ::= C[M_1 \ldots M_n] \]

**Expressions of predictable shape**
\[ e_i ::= \ldots \mid C^{pt}[e_1 \ldots e_n] \]

**Static free variables**
\[ SFV(T \prec \Phi) = \emptyset \]

**Well-formedness**

\[ \forall i \in \{1 \ldots n\}, \forall j \in \{1 \ldots n\}, \Gamma \vdash M_j^i \quad \forall i, j \in \{1 \ldots n\}, \Gamma \vdash C_i \neq C_j \]
\[ \Gamma \vdash (C_1[M_1^1 \ldots M_n^1] \ldots C_n[M_1^n \ldots M_n^n]) \quad \text{(WF-DATATYPE)} \]

\[ \frac{\Gamma \vdash t \quad \Gamma \vdash \Phi}{\Gamma \vdash (t \prec \Phi)} \quad \text{(WF-FOLD)} \]

\[ \forall D \in O, \Gamma \vdash D \quad \forall (t \prec \Phi), (t \prec \Phi') \in O, \Phi = \Phi' \]
\[ \forall D, D' \in O (DN(D) = DN(D') \lor DV(D) = DV(D')) \implies D = D' \quad \text{(WF-SIG')} \]

**Type strengthening**
\[ (t \prec \Phi)/p = (t \prec \Phi) \]

**Degree (for \( x \in FV(C^{pt}[e_1 \ldots e_n]) \))**

\[ Degree(x, C^{pt}[e_1 \ldots e_n]) = \bigwedge_{1 \leq i \leq n, x \in FV(e_i)} Degree(x, e_i) \]

---

**Figure 5.12:** Extension to datatypes

---

**Value:**
\[ v ::= \ldots \mid C^{pt}[v_1 \ldots v_n] \quad \text{Constructed value} \]
\[ \mid \text{match}^pt[v_1 \ldots v_n] \quad \text{Partial matching (for \( n \leq |\Phi| \))} \]

\[ \Phi = (C_1[M_1^1 \ldots M_n^1] \ldots C_n[M_1^n \ldots M_n^n]) \quad \text{(MATCH)} \]

\[ \text{match}^pt(C^{pt}[v_1 \ldots v_n]) v_1 \ldots v_n \rightarrow (v_1' \ldots v_n') \quad \text{(MATCH)} \]

---

**Figure 5.13:** Extending the dynamic semantics
functions, and returning a value of type \( t \). The matching function corresponding to the constructor \( C[M_1 \ldots M_n] \) expects \( n \) arguments of types \( M_1 \ldots M_n \), and returns a value of type \( t \). It appears here that the purpose of the \( pt \) annotation on the matching operator is to represent the type of the first argument to the matching. As syntactically, datatypes are not types, it could not be easily guessed otherwise.

The typing judgment for definitions has to be extended, because a single datatype definition corresponds to two declarations, an abstract type declaration and an unfolding. Thus, instead of a single declaration, the type of a definition is a finite set of declarations. To type a structure, successively type its definitions and take the (disjoint) union of the obtained signatures (rule TT-OUTPUT). Each datatype definition \( T \triangleright t = \Phi \) is checked correct, and its type is \( T \triangleright t : \star, t \prec \Phi \) (rule TT-DATATYPE).

By rule ST-SIG', signature matching now allows to forget some datatype declarations, only retaining an abstract type. Nevertheless, if the datatype is kept, rule ST-DATATYPE forces the two declarations to be equivalent. Two datatypes are equivalent if they define the same list of constructors, with equivalent types (rules DE-DATATYPE and DE-CON).

### 5.4 Examples

In this section, we give some example programs illustrating the use of mixin modules in some canonical situations. The calculus makes a syntactic difference between type and value names. Here, we do not syntactically distinguish between type and value identifiers, and prefer to prefix definitions and declarations with keywords \texttt{type} and \texttt{val} to disambiguate them. We syntactically distinguish names from variables, with the convention that variables begin with a lowercase letter, while names begin with an uppercase letter. Moreover, we assume that the language is extended with polymorphic comparison functions \( =, <, > \), some operations for booleans, such as prefix negation \texttt{not} and infix and operators, and a conditional construction \texttt{if then else}.

#### 5.4.1 Lists

We program a simple module implementing lists in \texttt{MML}. If we stick to monomorphic lists, that is, the type of elements is fixed to \texttt{int} for example, then it is straightforward. Let \( \Phi = \texttt{Nil}, \texttt{Cons}[\texttt{int}, t] \). We define the module \texttt{list} by

\[
\begin{align*}
o_{\text{list}} & \overset{=}{=} \text{def} \quad \text{type } T \triangleright t = \Phi \\
\text{val} \ & \text{Head} \triangleright \text{head} = \lambda x. \text{match}_{\Phi}[\text{int}] x \ \text{error} \\
\text{val} \ & \text{Tail} \triangleright \text{tail} = \lambda x. \text{match}_{\Phi}[t] x \ \text{error} \\
\text{val} \ & \text{Map} \triangleright \text{map} = \lambda f \lambda x. \text{match}_{\Phi}[t] x \ \texttt{Nil}^t [ ] \\
& \quad \lambda h d \lambda t. \texttt{Cons}^t [(f \ h d), (\text{map } f t l)]
\end{align*}
\]

and \texttt{list} = close \( \{ \emptyset ; o_{\text{list}} \} \).

Notice the use of \texttt{error}: we did not include exceptions in our formalism, but for sure they remain a useful construct in programming, and should be included in any practical application. The obtained module is of type \( \{ O \} \), where

\[
\begin{align*}
O & \overset{=}{=} \text{def} \quad \text{type } T \triangleright t : \star \\
& t \prec \texttt{Nil}, \texttt{Cons}[\texttt{int}, t] \\
\text{val} \ & \text{Head} \triangleright \text{head} : t \to \texttt{int} \\
\text{val} \ & \text{Tail} \triangleright \text{tail} : t \to t \\
\text{val} \ & \text{Map} \triangleright \text{map} : (\texttt{int} \to \texttt{int}) \to t \to t
\end{align*}
\]
Type path unfolding $\Gamma \vdash pt \prec \Phi$

$$(t \prec \Phi) \in \Gamma \quad \frac{\Gamma \vdash p : \{O\} \quad (T \triangleright t : M), (t \prec \Phi) \in O}{\Gamma \vdash p.T \prec \Phi \{ O \rightarrow p.O \} \quad (TU-PATH)}$$

$$(p.t) \equiv M 
\frac{\Gamma \vdash p \prec \Phi \quad \Gamma \vdash M \prec \Phi}{\Gamma \vdash pt \prec \Phi \quad (TU-EQ)}$$

Expressions

$$(C[M_1 \ldots M_n]) \in \Phi \quad \forall i \in \{1 \ldots n\}, \Gamma \vdash e_i : M_i \quad \frac{\Gamma \vdash C^{pt}[e_1 \ldots e_n] : pt}{\Gamma \vdash pt \prec \Phi \quad (TT-CONAPP)}$$

$$\Gamma \vdash \Phi \quad \frac{\Gamma \vdash pt \prec \Phi \quad \Gamma \vdash \Phi \equiv \Phi'}{\Gamma \vdash \text{match}^{pl}_{\Phi'} : \text{Match}(pt, \Phi) \quad (TT-MATCH)}$$

Definitions

$$\forall i \in \{1 \ldots n\}, \Gamma \vdash d_i : O_i \quad \frac{\Gamma \vdash (d_1 \ldots d_n) : (O_1 + \ldots + O_n)}{\Gamma \vdash \Phi \quad (TT-OUTPUT')}$$

$$\Gamma \vdash \Phi \quad \frac{\Gamma \vdash (T \triangleright t = \Phi) : (T \triangleright t : \ast, t \prec \Phi)}{\Gamma \vdash (T \triangleright t) : (T \triangleright t : \ast, t \prec \Phi) \quad (TT-DATATYPE)}$$

Declaration matching

$$\Gamma \vdash \Phi \equiv \Phi' \quad \frac{\Gamma \vdash (t \prec \Phi) \leq (t \prec \Phi')}{\Gamma \vdash (t \prec \Phi) \leq (t \prec \Phi') \quad (ST-DATATYPE)}$$

Signature matching

$$\forall 1 \leq i \leq n, \Gamma \vdash D_i \leq D'_i \quad \frac{\Gamma \vdash D_1 \ldots D_n, (t \prec \Phi)^* \leq D'_1 \ldots D'_n \quad (ST-Sig')}{\Gamma \vdash (T \triangleright t) : (T \triangleright t : \ast, t \prec \Phi) \leq (T \triangleright t) : (T \triangleright t : \ast, t \prec \Phi) \quad (ST-Sig')}$$

Datatype equivalence

$$\forall i \in \{1 \ldots n\}, \Gamma \vdash M_i \equiv M'_i \quad \frac{\Gamma \vdash C[M_1 \ldots M_n] \equiv C[M'_1 \ldots M'_n] \quad (DE\-CON)}{\Gamma \vdash (\phi_1 \ldots \phi_n) \equiv (\phi'_1 \ldots \phi'_n) \quad (DE\-DATATYPE)}$$

Type of match$^{pl}_{\Phi}$

$$\text{Match}(pt, \phi_1 \ldots \phi_n) = \forall t, pt \rightarrow \text{Constr}(t, \phi_1) \rightarrow \ldots \rightarrow \text{Constr}(t, \phi_n) \rightarrow t$$

$$\text{Constr}(t, C[M_1 \ldots M_n]) = (M_1 \rightarrow \ldots \rightarrow M_n \rightarrow t)$$

Figure 5.14: Extension of the typing judgments
Moreover, by type strengthening, at each place of use, the type declaration \( \text{type } T : t : * \) of \textit{list} becomes \( \text{type } T : t : \text{list}.T \).

\textbf{Parametric datatypes} This module is usable as an ML module on lists of integers. Notice however that the language does not feature parameterized datatypes, so it is not possible to implement directly a module dealing with lists of any type. We can try to encode parameterized datatypes though. A first attempt consists in adding a deferred, abstract \textit{elt} for the elements of the list. The corresponding mixin module \textit{openList} has the input signature \( I = \text{def type } \textit{Elt} : \text{elt} : * \) and the output is \( \text{elts} \), except that in the definition of \( T \), \textit{int} is replaced with \textit{elt} in the datatype. This mixin module indeed can produce a module for lists of any type, but it will generate different types at each instantiation, since our datatypes are generative. Moreover, the \textit{Map} function cannot be defined polymorphically.

This is not a problem if one wants to link with the open mixin module, but as argued by Szyiperski in [74], and discussed in section 2.3.3, it is sometimes more convenient to rely on a closed library module. To solve this issue, an extension of MML similar to Leroy’s [53] or Rusu’s [65] applicative functors, or Shao’s extended modules [71] seems possible although we have not formalized it. In Leroy’s vein, it could consist in giving type paths the grammar

\[
\text{pt} :: \quad \text{t} \mid \text{p}.T \\
| \quad [\text{p}_1 + \ldots + \text{p}_n].T
\]

where the production \( [\text{p}_1 + \ldots + \text{p}_n].T \) would denote the \textit{type component} \( T \) in any module computed by doing the sum \( \text{p}_1 + \ldots + \text{p}_n \). Then, a type \( M \) can be encapsulated in a mixin module \( \textit{eltMix} = \langle M \textit{ elt : elt } M \rangle \), and the type of lists with elements of type \( M \) is denoted by \( [\textit{openList} + \textit{eltMix}].T \).

The set of operations over lists can be extended polymorphically, as shown for instance by the following definition of the traditional functions \textit{foldLeft}, applying a function successively to all the elements of a list, and \textit{assoc}, looking for the element associated to a value in an association list.

We denote by \( \Phi_{\text{List}} \) the datatype \( \textit{Nil}, \textit{Cons}[ \textit{openList} + \textit{eltMix}].\textit{Elt} \), and by \( \text{pt} \) the type path \( [\textit{openList} + \textit{eltMix}].T \).

We define

\[
\text{let rec foldLeft} = \quad \lambda t. \lambda t\textit{Mix}.\lambda f.\lambda \textit{init}.\lambda l. \text{match pt} [t][l][i] \text{init} \\
\quad \lambda h.\lambda t. (\text{foldLeft} [t] \text{eltMix} f (f \text{ init } h) t l)
\]

and

\[
\text{let rec assoc} = \quad \lambda \text{eltMix}(\emptyset); \quad \text{type } \textit{Fst} : \textit{fst} : * ; \emptyset; \emptyset). \\
\text{type } \textit{Snd} : \textit{snd} : * \\
\text{type } \textit{Elt} : \textit{elt} : \textit{fst} \times \textit{snd} \\
\lambda v.\lambda l. \text{match pt} [\emptyset][\emptyset][\text{eltMix}, \text{Snd}][l] \text{error} \\
\quad \lambda h.\lambda t. \text{if } \text{ fst } h d = v \\
\quad \text{then } \text{ snd } h d \\
\quad \text{else } \text{ assoc eltMix } v t l
\]

Nevertheless, it is still not possible to easily define the polymorphic functions inside the mixin module for lists. Maybe, another solution is to define the parametric datatype as a mixin module \( \textit{val Lists-List} = \langle \text{type } \textit{Elt} : \textit{elt} : * ; \text{type } \textit{T} = \textit{Nil}, \textit{Cons}[\textit{elt}, t] \rangle \) and the \textit{type list} \( M \) is \( \textit{list + mixElt}.T \), for \textit{mixElt} a named mixin exporting the type \( M \). It is not obvious that this works in practice, because the argument \textit{mixElt} has to be named. In theory, all types could be wrapped in in mixin modules as their unique component \textit{Elt}, and referred to by the name of these mixin modules. For example, the module for lists and the \textit{Map} function would look like:
\[
\text{list} = \text{close}(\emptyset); \ \text{val} \ \text{List} \triangleright \ \text{list} = \langle \ Elt \triangleright \ \text{elt} : \star; \\
T \triangleright t = \text{Nil}, \text{Cons}[\text{elt}, t] \rangle
\]
\[
\text{val} \ \text{Map} \triangleright \ \text{map} = \lambda \text{mixElt} : \langle \emptyset; \ \text{Elt} \triangleright \ \text{elt} : \star; \emptyset \rangle. \\
\lambda t'. \lambda f : [\text{mixElt}], \text{Elt} \rightarrow t'. \\
\lambda \text{l} : [\text{mixElt} + \text{list}], T. \\
\text{match} \ [[\text{mixElt} + \text{list}] : T] \\
\text{Nil}, \text{Cons}[[\text{mixElt} + \text{list}] : \text{Elt}, [\text{mixElt} + \text{list}] : T] \rightarrow t'] \\
\text{Nil}[[\text{mixElt} + \text{list}] : T] \\
\lambda \text{hd}, \lambda \text{tl}. \text{Cons}[[\text{mixElt} + \text{list}] : T] ((f \ \text{hd}), (\text{map} \ \text{mixElt} [t'] f \ \text{tl})) \}
\]

This example fails to type-check, at least if \( f \) is given the type \([\text{mixElt}], \text{Elt} \rightarrow t'\), since its argument has type \([\text{mixElt} + \text{list}], \text{Elt}\). We made this mistake on purpose to show how subtle typing errors can appear with such encodings. One could envisage to introduce new type equations in the system, such as \([p + \ldots], T = [p], T\) if \([p], T\) is well-formed.

**Conclusion** On the whole, we arrive to the conclusion that this is both cumbersome and ad hoc, and typing these examples is not easy at all. Thus, the addition of primitive parameterized types would be beneficial. This could cause some difficulties, as shown by Harper et al. in [28]: [?] understand why. It is basically unification in the presence of higher-order, non-recursive type constructors with singleton kinds, which has been proved decidable by Chris Stone [72].

Notice though that the need for applicative mixin modules could be requested in practice, as applicative modules have proved useful.

In the remaining examples, for simplicity of the presentation, we assume that parameterized datatypes are primitive in the language, and that a module \text{list} has been defined, using them, with the following type:

\[
\text{list} : \{ \ \text{type} \ T \triangleright t : \ \lambda \text{elt} \ \text{list} . T(\text{elt}) \\
\emptyset. \text{Nil}, T \triangleright \text{Cons}[\text{elt}, t]\} \\
\text{val} \ \text{Head} \triangleright \text{head} : \forall \text{elt}. t[\text{elt}] \rightarrow \text{elt} \\
\text{val} \ \text{Tail} \triangleright \text{tail} : \forall \text{elt}. t[\text{elt}] \rightarrow t[\text{elt}] \\
\text{val} \ \text{Fold}_\text{Left} \triangleright \text{fold}_\text{Left} : \forall \text{elt}. t'. (t' \rightarrow \text{elt} \rightarrow t') \rightarrow t' \rightarrow t[\text{elt}] \rightarrow t' \\
\text{val} \ \text{Mem} \triangleright \text{mem} : \forall \text{elt}. t[\text{elt}] \rightarrow t[\text{elt}] \rightarrow \text{bool} \\
\text{val} \ \text{Max} \triangleright \text{max} : \forall \text{elt}. (\text{elt} \rightarrow \text{elt} \rightarrow \text{int}) \rightarrow t[\text{elt}] \rightarrow \text{elt} \\
\text{val} \ \text{Assoc} \triangleright \text{assoc} : \forall \text{fst}. \forall \text{snd}. \text{fst} \rightarrow t[\text{fst} \times \text{snd}] \rightarrow \text{snd} \}
\]

### 5.4.2 Simple interpreter

As shown by Duggan and Sourelis [31], mixin modules facilitate the modular development of compilers and, similarly, of interpreters. We illustrate it with a simple interpreter for a calculator [?] with variable bindings. It takes as arguments expressions consisting of operations on numbers, and possibly bindings of expressions, and returns the result if possible. We divide the implementation into three mixin modules.

**Evaluation mixin module** The first mixin \text{openEval} is in charge of the basic operations. It imports the type \text{env} of environments, the type \text{binding} of bindings, the type \text{variable} for expression variables, the function \text{findIn} \text{env}, which retrieves the value of a variable in an environment, and the function \text{bind}, which binds an expression to a variable in the environment.
\[
I_{\text{openEval}} = \text{def} \quad \text{type } Env \triangleright env : \star \\
\quad \text{type } Binding \triangleright binding : \star \\
\quad \text{type } Result \triangleright result : \text{int} \\
\quad \text{type } Variable \triangleright variable : \star \\
\quad \text{val } Find \_\text{in}\_\text{env} \triangleright find\_\text{in}\_\text{env} : \text{variable } \rightarrow \text{env } \rightarrow \text{result} \\
\quad \text{val } Bind \triangleright bind : \text{bindings } \rightarrow \text{env } \rightarrow \text{env}
\]

The mixin \texttt{openEval} must define the datatype \texttt{expr} of expressions, the type \texttt{result} for results of evaluation (integers), and the function \texttt{eval} which evaluates an expression in an environment. The datatype of expressions is defined as

\[
\Phi_{\texttt{Expr}} = \text{def} \quad \text{Var}[\text{variable}], \quad (*) \text{Variable} (*) \\
\quad \text{Plus}[\text{expr}, \text{expr}], \quad (*) \text{Addition} (*) \\
\quad \text{Const}[\text{int}], \quad (*) \text{Integer constant} (*) \\
\quad \text{Let}[\text{binding}, \text{expr}] \quad (*) \text{Let binding} (*)
\]

The output of the mixin module is as follows:

\[
\Phi_{\text{openEval}} = \text{def} \quad \text{type } Expr \triangleright expr : \Phi_{\text{Expr}} \\
\quad \text{type } Result \triangleright result' : \text{int} \\
\quad \text{val } eval = \lambda n_\text{env}. \lambda n_\text{expr}. \text{match}_{\Phi_{\text{Expr}}} [\text{result}] \_\text{expr} \\
\quad \lambda n. \text{find}\_\text{in}\_\text{env} n \_\text{env} \\
\quad \lambda n_\text{expr}_1. \lambda n_\text{expr}_2. (\text{eval }\_\text{expr}_1 \_\text{env} \_\text{expr}_2) + (\text{eval }\_\text{expr}_2 \_\text{env} \_\text{expr}_2) \\
\quad \lambda n. n \\
\quad \lambda \text{binding}. \lambda n_\text{expr}. \text{eval} (\text{bind }\_\text{binding} \_\text{an}\_\text{env} \_\text{an}\_\text{expr}) \_\text{an}\_\text{expr}
\]

and we can define \texttt{openEval} by \texttt{openEval} = \{\texttt{openEval}; \texttt{openEval}\}. It has type \{\texttt{openEval}; \texttt{openEval}; \texttt{openEval}; \texttt{openEval}\}, where

\[
\Phi_{\text{openEval}} = \text{def} \quad \text{type } Expr \triangleright expr : \star \\
\quad \text{expr } \sim \Phi_{\text{Expr}} \\
\quad \text{type } Result \triangleright result' : \text{int} \\
\quad \text{val } eval \triangleright eval = env \rightarrow expr \rightarrow result \\
\Phi_{\text{openEval}} = \text{def} \quad \{ \text{Eval } \circ \rightarrow \text{Eval} \\
\quad \text{Find}\_\text{in}\_\text{env } \circ \rightarrow \text{Eval} \\
\quad \text{Bind } \circ \rightarrow \text{Eval} \} \\
\Phi_{\text{openEval}} = \text{def} \quad \emptyset
\]

By rule ST-Sig', the implementation \texttt{IdExpr} can be hidden, by type constraint.

\textbf{Binding mixin module} The second mixin module deals with bindings. It imports the types of environments, variables, expressions and results, and the functions \texttt{Eval} and \texttt{Add to env}, which adds a variable and its value to the environment. Its import signature is thus

\[
I_{\text{openBind}} = \text{def} \quad \text{type } Env \triangleright env : \star \\
\quad \text{type } Variable \triangleright variable : \star \\
\quad \text{type } Expr \triangleright expr : \star \\
\quad \text{type } Result \triangleright result : \star \\
\quad \text{val } Add\_\text{to}\_\text{env} \triangleright add\_\text{to}\_\text{env} : \text{variable } \rightarrow \text{env } \rightarrow \text{env} \\
\quad \text{val } eval \triangleright eval : env \rightarrow expr \rightarrow result \\
\]

110
Given this, it can define the type $\text{Binding}$ of bindings, as association lists of variables and expressions, and the function $\text{bind}$ which takes a binding and an environment as arguments, evaluates the expressions, and binds the variables to the corresponding results in the environment.

$$o_{\text{openBind}} = \text{def type } \text{Binding} \triangleright \text{binding} = \text{list.}T[\text{variable} \times \text{expr}]$$
$$\text{val } \text{Bind} \triangleright \text{bind} = \lambda \text{bindings.} \lambda \text{env.} (\text{list.} \text{Fold.} \text{left} \ [\text{variable} \times \text{expr}] \ [\text{expr}] \ [\text{env}] \ \text{bind}_\text{one} \ \text{an.env} \ \text{bindings})$$
$$\text{val } \triangleright \text{bind}_\text{one} = \lambda \text{an.env.} \lambda \text{pair.} (\text{add.to.env} \ (\text{fst} \ \text{pair})) \ \text{eval an.env} \ (\text{snd} \ \text{pair}))$$

We can define the mixin module $o_{\text{openBind}} = \{ o_{\text{openBind}}; o_{\text{openBind}} \}$. The type $\text{Binding}$ can be made abstract by type constraint, which gives $o_{\text{openBind}}$ the type

$$o_{\text{openBind}} : \{ o_{\text{openBind}}; O_{\text{openBind}}; \rightarrow o_{\text{openBind}}; \rightarrow o_{\text{openBind}} \},$$

with

$$O_{\text{openBind}} = \text{def type } \text{Binding} \triangleright \text{binding} = \star$$
$$\text{val } \text{Bind} \triangleright \text{bind} : \text{bindings} \rightarrow \text{env} \rightarrow \text{env}$$

$$\rightarrow o_{\text{openBind}} = \text{def } \{ \text{Add.to.env} \odot \rightarrow \text{Bind}$$
$$\text{Eval} \odot \rightarrow \text{Bind} \}$$

$$\rightarrow o_{\text{openBind}} = \text{def Variable} \rightarrow \text{Binding}$$
$$\text{Expr} \rightarrow \text{Binding}$$

**Environment mixin module** The last mixin module we define handles environments. It has to define the type $\text{Env}$ of environments, and the functions $\text{Find.in.env}$ and $\text{Add.to.env}$ for finding and adding a variable binding in environments. It can be implemented by lists, as follows:

$$o_{\text{openEnv}} = \langle \text{type } \text{Variable} \triangleright \text{variable} : \star$$
$$\text{type } \text{Result} \triangleright \text{result} : \star$$
$$\text{type } \text{Env} \triangleright \text{env} : \text{list.}T[\text{variable} \times \text{result}]$$
$$; \text{Find.in.env} \triangleright \text{find.in.env} = \text{list.} \text{Assoc}[\text{variable}] \ [\text{result}]$$
$$\text{Add.to.env} \triangleright \text{add.to.env} = \lambda \text{res.} \lambda \text{an.env.} \text{Cons}[\text{env}][\text{v, res}, \text{an.env}] \rangle$$

Once again, the implementation of the type $\text{Env}$ can be hidden to the outside world by type constraint. Finally, the interpreter module is obtained by close($o_{\text{openEval}} + o_{\text{openBind}} + o_{\text{openEnv}}$).

**Comparisons** We think [?] that the example compiler sketched by Duggan and Sourelis in [31] is implementable in MML quite straightforwardly. However, Duggan and Sourelis [32] have proposed an extension of their initial language $DS$ with extensible datatypes and extensible constructors, which allows them to refine their interpreters incrementally. This is not possible in MML because datatypes are not extensible.

In [27, 29], Cray, Dreyer, Harper, and Puri investigate an extension of ML modules with recursive modules. They focus both on the possible type-theoretic definitions for such an extension, and on some example programs that should be encoded smoothly by recursive modules. As recursion was a primary concern in the design of mixin modules, MML encodes most of their examples quite smoothly, and our approach to datatypes allows to completely avoid the use of recursive types. Moreover, the problems recursive modules cause for separate compilation do not appear with mixin modules.
5.4.3 Bootstrapped data structures

Another class of examples Dreyer et al. use to demonstrate the expressive power of recursive modules in [29] are bootstrapped data structures, introduced by Okasaki [61]. The example they choose is the one of sets of sets, which is easily programmed in MML.

Sets of sets with mixin modules Sets of sets are built out of a mixin module openSet, implementing general sets. It imports the structure of the elements of the set: a type elt and a function elt_cmp : elt → elt → int, which compares two elements, returning 0 if they are equal, a positive integer if the first one is greater, and a negative integer otherwise. Given these, it defines the type of sets with elements of type elt (as lists), and some standard functionalities over sets. The mixin module could be constrained to hide the implementation of type set.

```
openSet = ⟨
  type Elt > elt :
    val Elt_cmp > elt_cmp : elt → elt → bool
  ;
  type Set > set = list.T[elt]
    val Empty > empty = Nil[Set[]]
    val Singleton > singleton = λx. Cons[elt, Nil[set][x]]
    val Cmp > cmp = λ1,2.
      match[elt]
        Nil,Cons[elt,elt] [int][l1]
        (match[elt]
          Nil,Cons[elt,elt] [int][l2]
          0
          (λhd,l. hd + l)
          1
          (λhd,l. elt_cmp (list.Max l1) (list.Max l2))))
    ...
⟩
```

After that, the mixin module for sets of sets wraps the one for sets. It defines the type of sets of sets relying on the imported type of sets, and forces the type elt to be itself.

```
openSos = ⟨
  type Elt > elt :
  type Set > set :
  type Sos > sos :
    sos < Int[int], Set[set]
    val Empty > empty : set
    val Singleton > singleton : elt → set
    val Set_cmp > set_cmp : set → set → int
    ;
    type Sos > sos' :
    sos' < Int[int], Set[set]
    type Elt > elt' = sos
    val Cmp > cmp = λsos1,2.[snipped code]
⟩
```

Finally, the two mixin modules can be merged together, redirecting the comparison functions to their expected names in each mixin module. The Cmp function of the openSet mixin module must be connected to the Set_cmp input of the openSos mixin module. Conversely, the Cmp function of the openSos mixin module must be connected to the Elt_cmp input of the openSet mixin module. The definitive comparison exported by the module Sos implementing sets of sets should be the one from openSos, so we rename Elt_cmp to Cmp in the obtained mixin module before to instantiate

112
it. Thus, $Sos$ is obtained by

$$ Sos = \text{close}( \text{openSet}[\text{Cmp} \mapsto \text{Set,\_}\text{cmp}] + $$

$$ \text{openSos}[\text{Cmp} \mapsto \text{Elt,\_}\text{cmp}] ]$$

*Sets of sets with recursive modules* In comparison, Dreyer et al. [29] implement recursive modules by a complicated elaboration process, transforming the original program into an expression of the underlying type theory. This theory features singleton kinds and *phase-splitting* rules [41], that separate modules into their static part and their dynamic part.

The source program for sets of sets resembles the following.

```plaintext
module type KEY = sig
  type key
  val compare : key -> key -> order
end

module type SET = sig
  type elt
  type set
  ...
end

functor MkSet(Key : KEY) = struct
  type elt = Key.key
  type set = M
  ...
end

signature SOS = sig rec Sos in
  type sos = Int of int | Set of Sos.SosSet.set
  module SosSet : SET with type elt = sos
end

module Sos = struct rec Sos : SOS in
  type sos = Int of int | Set of Sos.SosSet.set
  module SosKey = struct
    type key = sos
    let compare sos1 sos2 = ...
  end
  module SosSet = MkSet(SosKey)
end
```

The first module type **KEY** defines the signature of an ordered type: a type and a comparison function. The second module type **SET** defines the signature of a module implementing sets: the type **elt** of elements of the set, the type **set** of sets, and some functions over these types. The functor **MkSet** takes an ordered type as an argument, and returns a module, which we assume to implement sets. Formally, the functor **MkSet** is assumed to have the signature **functor** (**Key : KEY** -> **SET** with type **elt** = **Key.key**, although it is not its principal signature, since the implementation of the type of sets could be made manifest. The recursive module type **SOS** then defines the signature of a module implementing sets of sets: the type **sos** of sets of sets, and a sub-module implementing sets whose elements are of type **sos**. **SOS** is a *recursively dependent signature* (rds). The recursive module **Sos** implements the module type **SOS** in a straightforward way.

113
The dynamic part of the module is then elaborated to a transparent fixed-point, which does not allow dataflow definitions, since these are opaque but not flexible than the opaque fixed-point. The limited time-point requires the signature of the recursive module variable (here SOS) to be fully transparent, so dataflow definitions are elaborated by referring to their first elaboration in the StaticSos. We obtain something like:

\[
\text{module StaticSos} = \text{opaque struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{type sos = transparent struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{end end module sosSet = StaticSos.sosSet} \\
\text{end type set = \Int\ of \Int \text{ of set of Sos}.setSet} \\
\text{end type alt = sos \langle \text{Sos} \text{ of } \text{StaticSos} \rangle \text{in}} \\
\text{val Int : \Int \to sos} \\
\text{val Set : set \to sos} \\
\text{end end val expose : sos \to (\text{Int} + \text{set})} \\
\text{end}
\]

The elaboration of the module SOS is more complex and is done in two steps. First, the static part of the module is extracted as a set of definitions, possibly nested inside sub-modules. It is elaborated to an opaque fixed-point, which allows dataflow definitions (see §2 for details). We obtain something like:

\[
\text{module StaticSos} = \text{opaque struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{type sos = transparent struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{end end module sosSet = StaticSos.sosSet} \\
\text{end type set = \Int\ of \Int \text{ of set of Sos}.setSet} \\
\text{end type alt = sos \langle \text{Sos} \text{ of } \text{StaticSos} \rangle \text{in}} \\
\text{val Int : \Int \to sos} \\
\text{val Set : set \to sos} \\
\text{end end val expose : sos \to (\text{Int} + \text{set})} \\
\text{end}
\]

The program is therefore elaborated to the calculus as we explain informally. SOS is therefore elaborated into an opaque fixed-point that captures the use of recursive types. By phase-splitting in the calculus, the dynamic part of the recursive module is constrained to be abstract. In the proposed elaboration, it would then be impossible to extract the abstract part of it and put it in Static. To prevent such an issue, Dreyer et al. require the source recursive module not to export abstract types. This choice seems to be guided by two facts:

1. The dynamic part of the module is then elaborated to a transparent fixed-point, which does not allow dataflow definitions, since these are opaque but not flexible than the opaque fixed-point. The limited time-point requires the signature of the recursive module variable (here SOS) to be fully transparent, so dataflow definitions are elaborated by referring to their first elaboration in the StaticSos. We obtain something like:

\[
\text{module StaticSos} = \text{opaque struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{type sos = transparent struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{end end module sosSet = StaticSos.sosSet} \\
\text{end type set = \Int\ of \Int \text{ of set of Sos}.setSet} \\
\text{end type alt = sos \langle \text{Sos} \text{ of } \text{StaticSos} \rangle \text{in}} \\
\text{val Int : \Int \to sos} \\
\text{val Set : set \to sos} \\
\text{end end val expose : sos \to (\text{Int} + \text{set})} \\
\text{end}
\]

2. The elaboration of the module SOS is more complex and is done in two steps. First, the static part of the module is extracted as a set of definitions, possibly nested inside sub-modules. It is elaborated to an opaque fixed-point, which allows dataflow definitions (see §2 for details). We obtain something like:

\[
\text{module StaticSos} = \text{opaque struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{type sos = transparent struct rec \(\text{Sos} : \langle \text{Sos} / \text{StaticSos} \rangle \text{in}} \\
\text{end end module sosSet = StaticSos.sosSet} \\
\text{end type set = \Int\ of \Int \text{ of set of Sos}.setSet} \\
\text{end type alt = sos \langle \text{Sos} \text{ of } \text{StaticSos} \rangle \text{in}} \\
\text{val Int : \Int \to sos} \\
\text{val Set : set \to sos} \\
\text{end end val expose : sos \to (\text{Int} + \text{set})} \\
\text{end}
\]
• The first fact is that opaque fixed-points more or less encourage all references to other components of the module to be done through the recursive variable. For instance, consider the following recursive module.

module List =
opaque struct rec
List : sig rec List in
  type t = Nil | Cons of int * List.t
  val nthtail : List.t -> int -> List.t
end
in
type t = Nil | Cons of int * List.t
let nthtail (1 : List.t) n =
  if n = 0 then 1
  else match n with
  | Nil -> failwith 'list too short.'
  | Cons((hd : int), (tl : List.t)) -> nthtail tl (n - 1)
end

The components of this module contains a lot of references to other components through the recursive variable List, called module-recursive references by Dreger et al. Here, one could implement the type of List without any module-recursive reference. However, in the case of data-type definitions split across different sub-modules, module-recursive references are needed. Thus, it is simpler to consider a single data-type and to assume that the module-recursive reference in that type is needed. Then, in the body of nthtail, none of the module-recursive references could be turned into a local one (by eliminating the prefix List.); this would break the type-checking of the module. Indeed, in the pattern-matching, the second argument to Cons must be of type List.t, not t, so tl must have this type. Further, tl is given as an argument to nthtail in the recursive call, so the type of l has to be List.t too. Essentially, the problem is that it is impossible to unify t and List.t during type-checking.

• The second fact is that opaque fixed-points do not prevent the presence of equi-recursive type constructors. This is a problem because type-checking is not known to be decidable in the presence of higher-order equi-recursive type constructors.

These remarks lead Dreger et al. to prefer transparent fixed-points. Nevertheless, opaque fixed-points do not force all the type declarations to be transparent, which is sometimes convenient, as we have seen with the above example. Moreover, we think there are ways to work around the two problems of opaque fixed-points. For instance, elaborating all internal references into module-recursive references directly avoids the burden to write all module-recursive references by hand. Further, it is possible to modify the typing rule for opaque fixed-points in order to forbid equi-recursive type constructors and also to type-check the dynamic part of the module with all the information about the static part available. For reference, this leads to the following typing rule, with the notations of [29]:

\[
\begin{align*}
\Gamma \vdash S & \equiv [\alpha : \kappa, \sigma_1] \text{sig} & \Gamma[s \uparrow S] \vdash M & \equiv [\alpha, e] & \Gamma \vdash c \downarrow \kappa \\
\Gamma[s \uparrow [\alpha : s(c), \sigma_1]] \vdash e \downarrow \sigma_2 & \Gamma[\alpha : s(c)] \vdash \sigma_1 \equiv \sigma_2 [\alpha / (Fst s)] \text{type} \\
\Gamma \vdash \fix_S(s : S)M & : S
\end{align*}
\]

(We write \(\fix_S\) for “semi-transparent” fixed-point.) The rule forbids module-recursive references in the static part \(c\) of the module, thus relying on \(\text{ds}\)'s for static recursion. The dynamic part of the module is type-checked knowing the implementation of the static part. The obtained type for the dynamic part is checked equivalent to the expected type, knowing the implementation of the static part. This achieves the flexibility of transparent fixed-points, without forcing the user to
write a fully transparent signature. It is unclear whether it suffices for making the example of sets of sets work if \texttt{MkSet} returns an abstract type, because the underlying calculus used in \cite{29} does not feature generativity. It would be useful to try and transpose the discussion to the more recent formalism of \cite{28}.

In \texttt{MML}, modules components are mutually recursive by default, as well as signature components. Thus, the problems due to decoupling module-recursive and local references do not appear. Our way to work around recursive types is a bit cumbersome, as is the one for tracking ill-founded recursion: we keep static dependencies in the types of mixin modules. Dreyer et al. do not need such a machinery. Instead, one could argue that our way of dealing with recursive types is more orthogonal to design problems than theirs. As a result, the design of \texttt{MML} seems more natural than the one of \cite{29}. In particular, bundles of recursive modules are dealt with in a very \textit{ad hoc} way in \cite{29}, while they are encoded smoothly in \texttt{MML}.

5.4.4 Mathematical data structures

\textbf{Presentation}  In \cite{15, 14}, in the context of the \texttt{Foc} project \footnote{http://www-sop.inria.fr/~foc}, Boulmé et al. explore the implementation of a library of mathematical data structures dedicated to computer algebra, in OCaml. Let us first explain how they present computer algebra. Mathematical objects such as 1, or the polynomial $X^2 + X + 1$ are called \textit{entities}. In mathematics, entities are grouped in \textit{collections}, which express a link between these entities, possibly materialized by operations called \textit{methods}. For example, the entities $0, 1, 2, \ldots$ form the collection of natural numbers. Slightly more complex: the entities $0, 1, 2, \ldots$, together with the distinguished element $0$, the binary internal composition law $+$, and the unary internal composition law $-$, form the group of natural numbers. Collections have a \textit{carrier}, or \textit{representation type}. For natural numbers, it is \texttt{int}. Mathematical collections are in turn grouped by certain sets of properties, called \textit{species}. A species is a set of types and methods, which can be only declared, or defined, when common to all its collections. For example, the species of polynomials of one variable contains a default algorithm for multiplying polynomials, even if the carrier or the type of the coefficients are abstracted over. Species have \textit{interfaces}, specifying the set of methods they define. For more details, see \cite{15, 14, 62}.

The aim of the \texttt{Foc} project is to develop a certified library by extraction of OCaml programs from \texttt{Coq} specifications. \footnote{[\ldots]} (references) They have a precise list of criteria to be met by their implementation, insisting on incremental development, type abstraction, and code sharing. Essentially, for implementing such a library, objects do not offer enough abstraction mechanisms, whereas modules are not flexible enough with respect to incremental programming. As a result, they use a smart combination of objects and modules. A species is implemented by an abstract class, i.e. a class where some methods can be undefined. Interfaces are represented by class types. Collections are pairs of a type \texttt{t}, the carrier, and an object \texttt{meth}, containing the methods operating on \texttt{t}. When all methods of a species \texttt{s} are defined, it can be instantiated into a collection. For this, a module is created, which contains the corresponding carrier and the species \texttt{s}. For example, if the class \texttt{s} implements polynomials in one variable over real coefficients with lists of pairs of an integer and a floating point number (sparse representation), then the corresponding collection \texttt{poly} can be created by

\begin{verbatim}
module Poly = (struct
  type t = (int * float) list
  class meth = new s
end : sig
  type t
  class meth : st
end)
\end{verbatim}
where $\mathbf{s}$ is the type of $\mathbf{s}$, abstracted over the carrier. This achieves abstraction over the representation of the carrier. Extensibility and refinement are allowed by operating on the class $\mathbf{s}$.

MML allows a similar encoding of mathematical structures. Species can be encoded by mixin modules, abstract methods being represented by deferred components, and concrete methods with defined components. The carrier is represented by a type component. An interface is a module type. A collection is created by closing a mixin module, and immediately hiding the representation of the carrier.

**Simple examples** We show the idea by implementing the very beginning of the Foc library. For this we assume that MML has been extended with the overriding operator $\triangleright$ described in section [?], and with a macro expansion mechanism for abbreviating signatures. A module type can be included in a signature $I$ by a declaration of the form include $M$; if $M$ denotes the module type $\{O\}$, the signature $I$, include $M$ denotes the greatest lower bound $I'$ of $\{I\}$ and $\{O\}$, as module types. This means that forgetting some components is allowed. Moreover, we assume that an external and internal renaming and prefixing facility is given for signatures. The signature $I[(X \triangleright x) \Rightarrow (Y \triangleright y)]$ denotes $I$, with $X$ replaced by $Y$ and $x$ replaced by $y$, if it does not generate any conflict. The signature $I[(P \triangleright p) \cdot (X \triangleright x)]$ denotes $I$, with all the defined external and internal names prefixed by $P$ and $p$, respectively. We skip the details of this extension, although it is certainly non-trivial.$^2$

The minimal interface of species is defined as any printable carrier:

```
type basic_object_sig = {  type $T \triangleright t : \ast$,  val Print $\triangleright$ print : $t \rightarrow$ unit }
```

The basic species, at the top the semantic inheritance hierarchy of the structures we will define, is:

```
val basic_object =  {  include basic_object_sig  ;  $\varepsilon$  }
```

The interface of a set is defined by the following module type:

```
type set_sig = {  include basic_object_sig  ;  val Eq $\triangleright$ eq : $t \rightarrow t \rightarrow$ bool  ;  val Neq $\triangleright$ neq : $t \rightarrow t \rightarrow$ bool  }
```

The species of sets is the first to have a concrete method, $\text{Neq}$, which can be defined in terms of $\text{Eq}$:

```
val set = basic_object +  {  include set_sig  ;  val Neq $\triangleright$ neq = $\lambda x, y.$ not(eq $x$ $y$)  }
```

We define the interface of partial orders as:

```
type partial_order_sig = {  include set_sig  ;  val Leq $\triangleright$ leq : $t \rightarrow t \rightarrow$ bool  ;  val Lt $\triangleright$ lt : $t \rightarrow t \rightarrow$ bool  ;  val Geq $\triangleright$ geq : $t \rightarrow t \rightarrow$ bool  ;  val Gt $\triangleright$ gt : $t \rightarrow t \rightarrow$ bool  }
```

$^2$Lillibridge showed that it makes signature matching undecidable in OCaml [56]
Similarly to sets, only one of the four functions of `partial_order_sig` is needed to implement the three other ones. Thus, the species of partial orders can be defined as:

```plaintext
val partial_order = set + {
  include partial_order_sig
  ; val Lt >> lt = \x y. (leq x y) and (not (eq x y))
  ; val Geq >> geq = \x y. (leq y x)
  ; val Gt >> gt = \x y. (lt y x)
}
```

Lattices must match the same interface as partial orders, with two additional functions, the greatest lower bound and the least upper bound functions:

```plaintext
type lattice_sig = { include partial_order_sig
  ; val Glb >> glb : t -> t -> t
  ; val Lub >> lub : t -> t -> t }
```

The species of lattices does not have anything to define by default, and is therefore implemented as:

```plaintext
val lattice = partial_order + {
  include lattice_sig
  ; e
}
```

Then, the interfaces for mix- and max-lattices add the distinguished elements `Min` and `Max`, respectively:

```plaintext
type min_lattice_sig = { include lattice_sig
  ; val Min >> min : t
  ; val Is_min >> is_min : t -> bool }

val min_lattice = lattice + {
  include min_lattice_sig
  ; val Is_min >> is_min = \x. (eq x min)
}
```

```plaintext
type max_lattice_sig = { include lattice_sig
  ; val Max >> max : t
  ; val Is_max >> is_max : t -> bool }

val max_lattice = lattice + {
  include max_lattice_sig
  ; val Is_max >> is_max = \x. (eq x max)
}
```

The corresponding species can define the methods `Is_min` and `Is_max`, respectively, in terms of `Min` and `Max`:

```plaintext
val min_lattice = lattice + {
  include min_lattice_sig
  ; val Is_min >> is_min = \x. (eq x min)
}

val max_lattice = lattice + {
  include max_lattice_sig
  ; val Is_max >> is_max = \x. (eq x max)
}
```

Complete lattices can be implemented by inheriting both from min- and max-lattices.

```plaintext
type complete_lattice_sig = { include max_lattice_sig
  ; include min_lattice_sig }

val complete_lattice = max_lattice <~> min_lattice
```

The species of complete lattices has the mixin module `type`
\begin{align*}
\langle \text{type } T \triangleright t : \ast \rangle
\quad \text{; val } Neq \triangleright neq : t \to t \to \text{bool} \quad ; \to (\emptyset) \\
\text{val } Print \triangleright print : \text{unit} \\
\text{val } Eq \triangleright eq : t \to t \to \text{bool} \\
\text{val } Neq \triangleright neq : t \to t \to \text{bool} \\
\text{val } Leq \triangleright leq : t \to t \to \text{bool} \\
\text{val } Lt \triangleright lt : t \to t \to \text{bool} \\
\text{val } Geq \triangleright geq : t \to t \to \text{bool} \\
\text{val } Gt \triangleright gt : t \to t \to \text{bool} \\
\text{val } Glb \triangleright glb : t \to t \to t \\
\text{val } Lub \triangleright lub : t \to t \to t \\
\text{val } Min \triangleright min : t \\
\text{val } Is\_min \triangleright is\_min : t \to \text{bool} \\
\text{val } Max \triangleright max : t \\
\text{val } Is\_max \triangleright is\_max : t \to \text{bool}
\end{align*}

(We do not detail the dynamic dependencies, which are not interesting.)

It is then really easy to instantiate an example collection, with integers for examples. Let the complete lattice of natural numbers between 0 and 10 be implemented by the collection:

\begin{verbatim}
val open_int_lattice = complete_lattice + \{ \}
\quad \text{type } T \triangleright t = \text{int}
\quad \text{val } Print \triangleright print = print;nt
\quad \text{val } Eq \triangleright eq = \lambda x.\lambda y. (x = y)
\quad \text{val } Leq \triangleright leq = \lambda x.\lambda y. (x \leq y)
\quad \text{val } Glb \triangleright glb = \lambda x.\lambda y. \text{if } x \leq y \text{ then } x \text{ else } y
\quad \text{val } Lub \triangleright lub = \lambda x.\lambda y. \text{if } x \geq y \text{ then } x \text{ else } y
\quad \text{val } Min \triangleright min = 0
\quad \text{val } Max \triangleright max = 10
\}
\end{verbatim}

\text{val } int_lattice = close open_int_lattice

We can then decide that the algorithm for \text{Lt} is too inefficient, and incrementally implement an optimized collection \text{optimized_int_lattice}, with the comparison function from the library, as follows.

\begin{verbatim}
val open_optimized_int_lattice = open_int_lattice <- \{ includepartial_order_sig
\quad \text{val } Lt \triangleright lt = \lambda x.\lambda y. (x < y)
\}
\end{verbatim}

\text{val } optimized_int_lattice = close open_optimized_int_lattice

\textbf{Hard example (part VII): recursive polynomials, a first attempt} A very subtle example of a representation of mathematical structures is given in \cite{14} by recursive polynomials. It consists in representing polynomials in any number of variables, starting from a representation of polynomials in one variable, with natural degrees, parameterized over the type of their coefficients.

Polynomials introduce a slight complication in regard to the previous examples: they encapsulate a sub-structure of coefficients. A first natural attempt to represent such sub-structures is to wrap them as sub-modules. In this paragraph, we show how this strategy fails.

Define a module type for rings:
type ring_sig = { type T = t 
  val Eq = eq : t \rightarrow t \rightarrow bool
  val Zero = zero : t
  val Eq_zero = eq_zero : t \rightarrow bool
  val Un = un : t
  val Add = add : t \rightarrow t \rightarrow t
  val Minus = minus : t \rightarrow t \rightarrow t
  val Uminus = uminus : t \rightarrow t
  val Mult = mult : t \rightarrow t 
} 

The natural module type for polynomials has the ring of its coefficients as a virtual component, and some more functionalities related to polynomials:

```

type poly_sig = { val Coef = coef : ring_sig
  include ring_sig
  val Lift = lift : coef.T \rightarrow t
  val MultExtern = multExtern : coef.T \rightarrow t
  val Lc = lc : t \rightarrow coef.T
  val Iscoef = is_coef : t \rightarrow bool
}
```

The `lift` function lifts a coefficient to a polynomial of degree zero. The `multExtern` function multiplies a polynomial by a coefficient. The `lc` function returns the highest non-zero coefficient of a polynomial. The `is_coef` function checks if a polynomial is of strictly positive degree.

Some of these functions can be implemented in a generic way, in the following `poly` mixin module:

```
val poly = 
  include poly_sig
  ; val MultExtern = multExtern = \x. \p. (mult (lift c) p)
  val Eq_zero = eq_zero = \p. (coef.Eq_zero (lc p))
  val Iscoef = is_coef = \p. (eq p (lift (lc p)))
```

We can now define the mixin module of recursive polynomials. It relies on a representation of polynomials `Poly = my_poly` (the internal variable is for avoiding the conflict with `poly`). This sub-module defines polynomials in one variable, but this variable is unnamed. The idea is to use `my_poly` as a representation for polynomials in variable “X”, but also as a representation for polynomials in “Y”, and so on. Following this idea, a polynomial in “X” is a pair ("X", e), where e is of type `my_poly.T`. There remains a question though: what is the type of the coefficients? Semantically, one can see polynomials in variables “X_1”…“X_n” as polynomials in “X_1”, whose coefficients are polynomials in “X_2”...“X_n”, and so on. This is exactly how we proceed here. The coefficients of e are recursive polynomials. We maintain the invariant that the coefficients of a polynomial in a variable “X” are polynomials in variables inferior to “X”, according to the polymorphic comparison operators. Basic coefficients are imported as a `Base \rightarrow base` module. We obtain the (partially snipped) code, of figure 5.15, with \( \Phi = Base[base.T], Comp[string, my_poly.T] \).

The mixin module defines an intermediate type `support` as described above, and a sub-module `Rec_poly \rightarrow rec_poly`, defining the coefficients of the import module `my_poly`, i.e. the recursive polynomials. This is specified by the type sharing equation with `type Coef.T = support` in the expected type of `my_poly`. (Type sharing equations are not present in the language initially, but they are easily implemented using signature inclusion.) The sub-module `rec_poly` uses the generic module for polynomials `poly`, where `(Coef \rightarrow coef)` has been renamed to `(Base \rightarrow my_base)`, in order both to match the fact that it re-exports the imported module `Base \rightarrow base`, and to avoid conflict with its internal variable `base`. It specializes the type `T` of `poly` to `support`. The interesting functions are `Compose` and `Add`. 

120
\[
\text{val } \text{poly}_\text{rec} = \begin{cases} \\
\text{val } \text{Base } \triangleright \text{base : ring}_\text{sig} \\
\text{val } \text{Poly } \triangleright \text{my}_\text{poly} : \text{poly}_\text{sig} \text{ with type Coef.$T = \text{support}$} \\
\text{type } \text{Support } \triangleright \text{support : $^*$} \\
\text{support } \prec \Phi \\
; \text{type } \text{Support } \triangleright \text{support'} = \text{!!} \\
\text{support'} \prec \Phi \\
\text{val } \text{Rec}_\text{poly } \triangleright \text{rec}_\text{poly} = \text{close} ( \\
\text{poly}[(\text{Coef } \triangleright \text{coef}) \rightarrow (\text{Base } \triangleright \text{my}_\text{base})] \leftarrow \\
(\text{include } \text{poly}_\text{sig}[(\text{Coef } \triangleright \text{coef}) \rightarrow (\text{Base } \triangleright \text{my}_\text{base})]) \\
\text{with type } T = \text{support} \\
\text{and type } \text{Base}.T = \text{base}.T \\
\text{val } \text{Compose } \triangleright \text{compose : string } \rightarrow \text{my}_\text{poly}.T \rightarrow t \\
; \text{val } \text{Base } \triangleright \text{my}_\text{base'} = \text{base} \\
\text{type } T \triangleright t' = \text{support} \\
\text{val } E_q \triangleright eq' = \lambda x.\lambda y.(x = y) \\
\text{val } \text{Zero } \triangleright \text{zero'} = \text{Base}'[\text{base}.\text{Zero}] \\
\text{val } \text{Un } \triangleright \text{un'} = \text{Base}'[\text{base}.\text{Un}] \\
\text{val } \text{Compose } \triangleright \text{compose'} = \lambda w.\lambda. \\
\text{if } \text{my}_\text{poly}.\text{Is}_\text{coef} \text{l then my}_\text{poly}.\text{Lcl} \\
\text{else Comp}_\text{t}[w,l] \\
\text{val } \text{Lift } \triangleright \text{lift'} = \lambda a.\text{Base}'[a] \\
\text{val } \text{Lc } \triangleright \text{lc'} = \lambda x.\text{match}_\text{t}^\text{base}[\text{base}.T]x \\
\lambda a.\lambda w.\lambda.(\text{lc } (\text{my}_\text{poly}.\text{Lcl})) \\
\text{val } \text{Add } \triangleright \text{add'} = \lambda x_1.\lambda x_2.\text{match}_\text{t}^\text{base}[t]x_1 \\
\lambda a_1.\lambda a_2.\text{match}_\text{t}^\text{base}[t]x_2 \\
\lambda a_2.\text{Base}_\text{base}.T[\text{base}.\text{Add } a_1 a_2] \\
\lambda a_2.\lambda x_2. (\text{compose } v_2 (\text{my}_\text{poly}.\text{Add } (\text{my}_\text{poly}.\text{Lift } x_1) l_2)) \\
\lambda v_1.\lambda x_1.\text{match}_\text{t}^\text{base}[t]x_1 \\
\lambda v_2. (\text{compose } v_1 (\text{my}_\text{poly}.\text{Add } (\text{my}_\text{poly}.\text{Lift } x_2) l_2)) \\
\lambda v_2. (\text{compose } v_2 (\text{my}_\text{poly}.\text{Add } (\text{my}_\text{poly}.\text{Lift } x_1) l_2)) \\
[\ldots \text{snipped} \ldots]
\end{cases}
\]

Figure 5.15: Recursive polynomials (first attempt)
Compose takes a variable \( v \) and a polynomial \( l \) (of type \( my\_poly.T \)), and returns the same polynomial, seen as a polynomial in \( v \), in canonical form (of type \( support \)). The variable \( v \) is assumed superior to the variables used in the coefficients of \( l \). If \( l \) is of degree zero, then the function returns the corresponding coefficient, which is indeed of type \( support \). If \( l \) is of strictly positive degree, then the function returns \( \text{Comp}^l[v,l] \).

Add takes two recursive polynomials \( x_1 \) and \( x_2 \) of type \( t \), and returns their sum.

- If both arguments are base coefficients, then the sum is the sum of these coefficients.
- If both arguments are composed polynomials, i.e. constructed with the \( \text{Comp} \) constructor, then the variables are examined.
  - If both \( x_1 \) and \( x_2 \) are recursive polynomials in the same variable \( v \), then the underlying polynomials are summed, and the result \( l \) is injected into recursive polynomials in \( v \) by the \( \text{compose} \) function.
  - Otherwise, the argument with the greatest variable, say \( x_1 \) for example, is decomposed into the variable \( v \) and the underlying polynomial \( l \). The coefficients of \( l \) are recursive polynomials in variables inferior to \( v \), so \( x_2 \) is semantically of the same class them. Therefore, it is lifted by \( my\_poly.Lift \) to a polynomial of degree zero, and added to \( l \). The result is then injected back into recursive polynomials in \( v \) by the \( \text{compose} \) function.
- If one argument, say \( x_1 \) is a base coefficient, and the other is a recursive polynomial \( v,l \), then \( x_1 \) is semantically in the same class as coefficients of \( l \) since all its variables are inferior to theirs. So, it can be lifted by \( my\_poly.Lift \) to a polynomial of degree zero, and added to \( l \). The result is then injected back into polynomials in \( v \) by the \( \text{compose} \) function.

Until now, no problem arose. But assume now that we have implemented the ring of integers \( \text{int}\_ring \) and a mixin module for sparse polynomials \( \text{sparse}\_poly \). If we try to construct recursive polynomials by composing these two mixin modules with \( \text{poly}\_rec \), we write

\[
\text{val try} = \text{poly}\_rec + \\
\{ \text{val Rec}\_poly \triangleright \text{rec}\_poly : \text{poly}\_sig \\
\text{val Poly} \triangleright \text{my}\_poly = \text{close} ( \text{sparse}\_poly + \\
\text{val Base} \triangleright \text{base} = \text{int}\_ring ) \}
\]

Unfortunately, this expression is ill-typed, since there is a dependency cycle between \( \text{rec}\_poly \) and \( \text{my}\_poly \), and both are expressions of the shape \( \text{close} \ldots \), which are considered of unpredictable shape. In fact, it would be very difficult to let the system accept this. A solution could be to rely on types to guess the shape of both modules. But then, one has to check that one does not try to inspect the value of the other before it has been defined. And in this particular case, it is far from obvious. Indeed, the components of each module can be considered safe from their definitions, but what about the components of \( \text{sparse}\_poly \)? They perfectly could require some components of \( \text{rec}\_poly \). Thus, the dependency analysis must be refined if we want to allow this example to be well-typed.

Hard example (part VII): recursive polynomials, a solution There is a different solution to implement recursive polynomials, using roughly the same idea, but flattening all the sub-modules. The problems of name conflicts are solved by prefixing the names, reproducing in a flat way the namespace separations induced by module boundaries in the first attempt.

The \( \text{ring}\_sig \), \( \text{poly}\_sig \) module types, and the \( \text{poly} \) and mixin module are defined as above, except that the sub-module representing coefficients is now inlined in \( \text{poly}\_sig \) (and consequently also in \( \text{poly} \)). The modified module type is
type poly_sig = {  inducting_sig[(Coef >> coeF .) : (X >> x)]
  inducting_sig
  val Lift >> lift : coeF -> t
  val Mult_Extern >> mult_extern : coeF -> t -> t
  val Le >> le : t -> coeF
  val Is_Coef >> is_Coef : t -> bool }

Coefficients are represented by the included signature ring_sig[(Coef >> coeF .) : (X >> x)], which brings the type Coef.T of coefficients, and ring operations on it, such as Coef.Mult and Coef.Add. Polynomials are represented by the second included ring signature (without prefixing). The new mixin module for recursive polynomials is presented in figure 5.16.

As in the first attempt, the mixin module bases on the generic mixin module for polynomials, but here, the renaming of Coef to Base must be done component-wise. Indeed, it would otherwise modify all the names. For readability, as a shorthand, we write only the names in the renaming, not the variables. They are renamed accordingly. The base coefficients of our recursive polynomials are imported as a ring_sig signature, prefixed with Base_, to mimic the imported Base sub-module of the first attempt. Similarly, the sub-module Poly of the first attempt is imported here as a poly_sig signature. The type sharing equation Coef.T = support is converted into a renaming removing the prefix of all the components beginning with Poly_Coef_. this makes them match the components corresponding to recursive polynomials. The main data-type is then defined, but must be modified according to the new naming conventions: Φ’ = Base[base_], Comp[ring, poly_]. The rest of the mixin module is defined similarly, only replacing some accesses to sub-modules with direct accesses to prefixed components of the main mixin module.

This second attempt is successful, since a module of recursive polynomials can be built on the ring of integers int_ring and a mixin module for sparse polynomials sparse_poly (which has been flattened to match the signature poly_sig). The code is as follows:

val int_recur_sive_polynomials = (int_ring[(Base_ >> base_). (X >> x)]
+ (sparse_poly [(Poly_ >> poly_). (X >> x)]
| type Poly_Coef.T >>> T
  val Poly_Coef.Eq <<< E_q Poly_Coef.Zero <<< Zero
  Poly_Coef.Eq.zero <<< E_q.zero Poly_Coef.Un <<< Un
  Poly_Coef.Add <<< Add Poly_Coef.Minus <<< Minus
  Poly_Coef.Uminus <<< Uminus Poly_Coef.Mult <<< Mult
| rec_poly_flat));;
val rec_poly_strat =
  poly | type Coef_T      → Base_T
        val Coef_Eq       → Base_Eq  Coef_Zero    → Base_Zero
        Coef_Eq_zero    → Base_Eq_zero Coef_Un      → Base_Un
        Coef_Add        → Base_Add  Coef_Minus   → Base_Minus
        Coef_Uminus     → Base_Uminus Coef_Mult     → Base_Mult

⟨ include ring_sig[Base_→ base_] · (X > x) ⟩
include(poly_sig with type Coef_T = support
[(Poly_→ poly_), (X > x)]
[ type Poly_Coef_T → T
  val Poly_Coef_Eq → Eq       Poly_Coef_Eq_zero → Zero
  Poly_Coef_Add → Add       Poly_Coef_Minus → Minus
  Poly_Coef_Uminus → Uminus  Poly_Coef_Mult → Mult ]

val Lift ∘ lift : base_t → t
val Mult_External ∘ mult_extern : base_t → t → t
val Lc ∘ lc : t → base_t
val Is_Coef ∘ is_coef : t → bool
val Compose ∘ compose : string → poly_t → t

type Support ∘ support : ⋆
support < @ Φ'
  type Support ∘ support' : ⋆
support' < @ Φ' type T > t' = support
val Eq ∘ eq' = λx.λy.(x = y)
val Zero ∘ zero' = Base'[base_zero]
val Un ∘ un' = Base'[base_un]
val Compose ∘ compose' = λv.λl.
  if poly_is_Coef l then poly_Lc l
  else Comp'(v,l)
val Lift ∘ lift' = λa.Base'[a]
val Lc ∘ lc' = λx. match_T[base_T] x
λa.a
λv.λl. (lc (poly_Lc l))
val Add ∘ add' = λx₁.λx₂. match_T[+] x₁
λa₁. match_T[+] x₂
  λa₂. Base_T[base_+] [base_add a₁ a₂]
  λv₁, λl₁. (compose v₁ (poly_add (poly_Lift x₁) l₂))
  λv₁, λl₁, match_T[+] x₂
  λv₂. (compose v₂ (poly_add (poly_Lift x₂) l₁))
  λv₂, λl₂, if v₁ = v₂ then (compose v₂ (poly_add l₁ l₂))
  else if v₁ > v₂
  then (compose v₂ (poly_add (poly_Lift x₂) l₁))
  else (compose v₂ (poly_add (poly_Lift x₁) l₂))
[... snipped ... ]

Figure 5.16: Flattened recursive polynomials

124
Part III

Compilation of mixin modules
Chapter 6

Typed compilation without local definitions

6.1 Intuitions

In this chapter, we present an efficient compilation scheme for a subset of MM. Let us first give intuitions on it. 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 ???. These two structures are translated to:

\[
\begin{align*}
M1 &= \{ f = \lambda g. \lambda x. \ldots g \ldots; u = \lambda f. f \ 0 \} \\
M2 &= \{ g = \lambda f. \lambda x. \ldots f \ldots; v = \lambda g. g \ 1 \}
\end{align*}
\]

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):

\[
\begin{align*}
&\text{let rec } f = m.f \ g \ \text{and } g = m.g f \ \text{in} \\
&\text{let } u = m.u f \ \text{in} \\
&\text{let } v = m.v g \ \text{in} \\
&\{ f = f; g = g; u = u; v = v \}
\end{align*}
\]

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 \( \lambda \)-calculi.

In fact, the let rec of MM is almost powerful enough to model such fixpoints. We choose as the target language of our compilation scheme the \( \lambda \) calculus, featuring a let rec construct that slightly extends that of MM. It allows to group all the components within a single binding:
\[
x \in \text{Vars} \quad \text{Variable}
\]
\[
X \in \text{Names} \quad \text{Name}
\]
Expression: \[ e ::= x \]
\[
\{ X_1 = e_1 \ldots X_n = e_n \} \quad \text{Record}
\]
\[
e . X \quad \text{Record selection}
\]
\[
\text{let rec } x_1 = e_1 \ldots x_n = e_n \text{ in } e \quad \text{let rec}
\]
\[
( X_1 \mapsto x_1 \ldots X_n \mapsto x_n ; d_1 \ldots d_m ) \quad \text{Structure}
\]
\[
e_1 + e_2 \mid \text{close } e \quad \text{Composition, closure}
\]
\[
ed_1 \ldots d_n \mid e \quad \text{Projection, deletion}
\]
\[
e [ X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n ] \quad \text{Renaming}
\]
\[
ed_{X > Y} \quad \text{Splitting}
\]

Definition: \[ d ::= X [ x_1 \ldots x_n ] \triangleright x = e \quad \text{Named definition} \]

Figure 6.1: Syntax of \( \text{MM}_c \)

\[
\text{let rec } f = m . f \ g
\]
\[
g = m . g \ f
\]
\[
u = m . u \ f
\]
\[
v = m . v \ g
\]
\[
in \{ f = f ; \ g = g ; \ u = u ; \ v = v \}
\]

We have not proven any encoding property of our compilation scheme. We would at least like to have a (weaker) soundness result for it, and a simple idea to show it is to set up a sound type system for \( \lambda_c \), and show that the expressions generated by our compilation scheme are well-typed. However, the type system of \( \text{MM} \) would not accept them, so we have to find a finer type system. Fortunately, Bondol [13] has already developed a non-standard type system for a call-by-value calculus that supports such single recursive definitions. Later, we have extended it to mutually recursive definitions in [46]. Here, we adapt the ideas of [46] to \( \lambda_c \), and our result is that the compiled terms are well-typed.

# 6.2 Definition of the compilation scheme

## 6.2.1 Restricting the source language: \( \text{MM}_c \)

The syntax of \( \text{MM}_c \) terms and types is defined in figure 6.1. The meaning of meta-variables is kept from the presentation of \( \text{MM} \) (section 3.1). The language is the same, except that anonymous definitions have dissapeared, and the freezing, hiding, and showing operations, that were using them. The operations on the structure of expressions are defined by restriction of the ones of \( \text{MM} \). The notion of syntactic correctness is maintained identical as for \( \text{MM} \), and expressions are similarly identified modulo correct variable renaming.

The operational semantics are defined exactly as for \( \text{MM} \), without the contraction rules \text{FREEZE}, \text{HIDE}, \text{SHOW}, and letting the meta-variable \text{op} range over the restricted set of operators (see figure 3.2), and denote by \text{op}[e] the application of \text{op} to the expression \( e \). The syntax for contexts is modified accordingly. Also, the notions of predictable shape and of degree remain the same. In particular, the \text{Degree} function returns \( \emptyset \) on all kinds of expressions, except on mixin modules and records, where it returns \( \emptyset \).

The definition of the type system slightly differs from that of \( \text{MM} \). Indeed, the output sections of mixin module types are now lists of types, indexed by names, as indicated in figure 6.2. They
\[
M \in \text{Types} \quad ::= \quad \{O\} \mid \langle I; O; G \rangle \\
I \quad \in \quad \text{Names} \quad \text{\(\text{Fin} \rightarrow\) Types} \\
O \quad ::= \quad \epsilon \mid X \mapsto M, O \\
G \quad \in \quad \text{Fin} \{X \xrightarrow{\chi} Y \mid X, Y \in \text{Names}, \chi \in \text{Degrees}\} \\
\Gamma \quad \in \quad \text{Vars} \quad \text{\(\text{Fin} \rightarrow\) Types}
\]

Figure 6.2: Types for \(\text{MM}_e\)

\[
\begin{align*}
\chi &= \text{Degree}(x', e) \quad (X', x') \in \text{dom}(\langle i; o \rangle) \quad (X[x^*] \triangleright x = e) \in o \\
X' \xrightarrow{i; o} X \\
(X_i, x_i) &\in \text{dom}(\langle i; o \rangle) \quad (X[x_1 \ldots x_n] \triangleright x = e) \in o \\
X_i \xrightarrow{\bigcirc} X
\end{align*}
\]

Figure 6.3: Dependencies in a \(\text{MM}_e\) structure

are still supposed to be finite maps. Thus, in the following, the meta-variable \(I\) still denotes a finite map from names to types, but the meta-variable \(O\) now denotes a list of types indexed by distinct names. The typing rules are modified accordingly: for a mixin module \(\langle i; o \rangle\), the output section of the result type preserves the order in which the components appeared in \(o\). This does not change the typing rule T-STRUCT however. The meaning of the rule T-SUM slightly changes though, because we have to define the disjoint union operation \(\uplus\) on indexed lists. It is defined, if the two lists define disjoint sets of names, as their concatenation, and undefined otherwise. Thus, there is an implicit side-condition in rule T-SUM from the point of view of this section, requiring that the output sections of the two summed mixin modules define disjoint sets of names.

The notion of graph and the corresponding operations are greatly simplified by the absence of local definitions: all the considered graphs are abstract (i.e. graphs on names only). The way to compute the dependency graph \(\rightarrow_{i; o}\) of a structure \(\langle i; o \rangle\) is also simpler, as described in figure 6.3: nodes are simply names, and no lift operation is necessary.

Our goal is to translate well-typed terms of \(\text{MM}_e\) 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. Fortunately, proposition 7 remains true.

**Proposition 8 (Types well-formed)** If the types in \(\Gamma\) are well-formed, and \(\Gamma \vdash e : M\), then \(M\) is well-formed.

### 6.2.2 The target language \(\lambda_c\)

The target language for our translation is the \(\lambda_c\) calculus, a variant of the \(\lambda\)-calculus with records and recursive definitions introduced by Boudol [13].

**Syntax**

The syntax of \(\lambda_c\) is defined in figure 6.4. Intuitively, it is a subset of \(\text{MM}_e\), where mixin module constructs have been replaced by functions and applications, and the let rec has been extended (see below) The meta-variables \(X\) and \(x\) range over names and variables, respectively. Variables are used as binders, as usual. Names are used for accessing record fields, as an external
\[
x \in \text{Vars} \quad \text{Variable}
\]
\[
X \in \text{Names} \quad \text{Name}
\]
\[
\phi := =[n] \mid =[?]
\]
Expression:
\[
e \in \text{expr} ::= x \mid \lambda x.e \mid e_1 e_2
\]
\[
\mid \{X_1 = e_1 \ldots X_n = e_n\}
\]
\[
\mid e.X
\]
\[
\mid \text{let rec } x_1 \phi_1 e_1 \ldots x_n \phi_n e_n
\]
\[
\text{in } e
\]

Figure 6.4: Syntax of $\lambda_c$

- More meta-variables:
  \[
s ::= X_1 = e_1 \ldots X_n = e_n \quad \text{Record}
\]
  \[
b ::= x_1 \phi_1 e_1 \ldots x_n \phi_n e_n \quad \text{Binding}
\]

- Notations:

  For a finite map $f$, and a set of variables $P$,

  \text{dom}(f) \text{ is its domain,}
  \text{cod}(f) \text{ is its codomain}
  \text{f}_P \text{ is its restriction to P,}
  \text{and f}_{/P} \text{ is its restriction to dom}(f) \setminus P.

- Expressions of predictable shape:
  \[
e_\downarrow \in \text{Predictable ::= } \{o\} \mid \lambda x.e \mid \text{let rec } b \text{ in } e_\downarrow
\]

Figure 6.5: Meta-variables and notations
interface to other parts of the expression. Figure 6.5 recapitulates the meta-variables and notations we introduce in the remainder of this section. The syntax includes the λ-calculus constructs; variables \( x \), abstraction \( \lambda x.e \), and application \( e_1 e_2 \). The language also includes records \( \{X_1 = e_1 \ldots X_n = e_n\} \), record selection \( e.X \) and a let rec construct. A mutually recursive definition has the shape let rec \( x_1 \circ e_1 \ldots x_n \circ e_n \) in \( e \), where arbitrary expressions are syntactically allowed as the right-hand side of a definition.

**Syntactic correctness** Records \( s = (X_1 = e_1 \ldots X_n = e_n) \) and bindings \( b = (x_1 \circ e_1 \ldots x_n \circ e_n) \) are required to be finite maps: a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice.

In a let rec binding \( b = (x_1 = e_1 \ldots x_n = e_n) \), we say that there is a forward reference from \( x_i \) to \( x_j \) if \( 1 \leq i \leq j \leq n \) and \( x_j \in \text{FV}(e_i) \). A forward reference from \( x_i \) to \( x_j \) is syntactically forbidden, except when \( e_j \) is of predictable shape. An expression of predictable shape is a record, a function, or a binding followed by an expression of predictable shape. Formally \( e \downarrow \in \text{Predictable} := \{ s \} | \lambda x.e | \text{let rec } b \text{ in } e \downarrow \).

**Sequences** Records and bindings are often considered as finite maps in the sequel. We refer to them collectively as sequences, and use the usual notions on finite maps, such as the domain \( \text{dom} \), the codomain \( \text{cod} \), the restriction \( \gamma_P \) to a set \( P \), or the co-restriction \( \gamma_P^\perp \) outside of a set \( P \).

**Structural equivalence** We consider the expressions equivalent up to alpha-conversion of binding variables in structures and let rec expressions. The set of terms of \( \lambda_c \) is defined as the set of structural equivalence classes.

**Semantics**

The semantics of \( \lambda_c \) is quite similar to that of \( \text{MM}_c \), except for what concerns let rec bindings. A first difference is that a binding defining only values is considered fully evaluated only if these values match the corresponding size indications: if a value of size \( n \) is expected (annotation \( [n] \)), then the defined value must have this size; if a value of unknown size is expected (annotation \( [\_] \)), then any value will do. From now on, the meta-variable \( b_v \) for bindings of \( \lambda_c \) denotes such fully evaluated bindings. This implicitly appears in the definition of results and evaluation contexts.

As shown in figure 6.6, values include functions \( \lambda x.e \) and records of values \( \{s_v\} \), where \( s_v \) denotes an evaluated record \( X_1 = v_1 \ldots X_n = v_n \).

The semantics of record selection and of function application are defined in figure 6.7, by computational contraction rules, defining the local computational contraction relation \( \rightsquigarrow_c \). Record projection selects the appropriate field in the record; and the application of a function \( \lambda x.e \) to a value \( v \) reduces to the body of the function, where the argument has been bound to \( x \) by let rec.

In \( \lambda_c \), for mostly technical reasons, we distinguish the topmost binding syntactically: the global computational reduction relation \( \rightsquigarrow_c \), is a binary relation on configurations \( c \), which are pairs of a binding, the topmost binding, and an expression, written \( b \vdash e \) (see figure 6.6). Here, the topmost binding is close to the usual notion of runtime environment, with the additional feature that bound values can be mutually recursive.

The rules for handling let rec and the notion of evaluation contexts are adapted to this notion of configuration. The computational contraction rule LIFT remains the same. The computational
Figure 6.6: Configurations and answers in $\lambda_c$

- **Computational contraction rules**

\[
\begin{align*}
\{X_1 = v_1 \ldots X_n = v_n\}.X_i & \leadsto_c v_i \quad \text{(PROJECT)} \\
\frac{x \notin FV(v)}{\lambda x.e \leadsto_c \text{let rec } x = v \text{ in } e} \quad \text{(BETA)}
\end{align*}
\]

\[
\frac{\text{dom}(b) \perp FV(L)}{L \left[\text{let rec } b \text{ in } e\right] \leadsto_c \text{let rec } b \text{ in } L [v]} \quad \text{(LIFT)}
\]

- **Computational reduction rules**

\[
\frac{e \leadsto_c e'}{E [e] \leadsto_c E [e'] \quad \text{(CONTEXT)}}
\]

\[
\frac{\text{dom}(b_1) \perp \{x\} \cup \text{dom}(b_v, b_2) \cup FV(b_v, b_2) \cup FV(f)}{(b_v, x = (\text{let rec } b_1 \text{ in } e), b_2 \vdash f) \leadsto_c (b_v, b_1, x = e, b_2 \vdash f) \quad \text{(IM)}}
\]

\[
\frac{\text{dom}(b) \perp (\text{dom}(b_v) \cup FV(b_v))}{(b_v \vdash \text{let rec } b \text{ in } e) \leadsto_c b_v, b \vdash e \quad \text{(EM)}}
\]

\[
\frac{E [N](x) = v}{E [N [x]] \leadsto_c E [N [v]]} \quad \text{(SUBST)}
\]

- **Evaluation contexts**

- Lift context:
  \[
  L ::= \square e \mid e \square \mid \square .X \mid \{ S \}
  \]

- Nested lift context:
  \[
  F ::= \square \mid L [F]
  \]

- Evaluation context:
  \[
  E ::= (b_v \vdash F) \mid (B \vdash F) \vdash e
  \]

- **Access in evaluation contexts**

\[
\begin{align*}
(b_v \vdash F)(x) &= b_v(x) \quad \text{(EA)} \\
(b_v, y = F, b \vdash e)(x) &= b_v(x) \quad \text{(IA)}
\end{align*}
\]

Figure 6.7: Reduction semantics for $\lambda_c$

132
reduction relation extends the computational contraction relation to any evaluation context $E$, as defined in figure 6.7. An evaluation context $E$ is a nested lift context, either inside a partially evaluated binding, or under a fully evaluated binding. The reduction rules are modified accordingly.

The target language

The computational reduction relation on expressions is compatible with structural equivalence. Hence we can define computational reduction over equivalence classes of expressions, obtaining the reduction relation $\rightarrow$.

**Definition 18** The $\lambda_c$ language is the set of terms, equipped with the relation $\rightarrow$.

$\lambda_c$ features a let rec that is slightly extended over the ones of ML or OCaml. We will now show how to compile it. Our target language for this compilation is presented in the next section and is a $\lambda$-calculus without a let rec at all, but with notions of heap, and locations.

### 6.2.3 Compilation scheme

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

The translation scheme for our language is defined in figure 6.8. The translation is type-directed and operates on terms annotated by their types. For the core language constructs (variables, constants, abstractions, applications), the translation is a simple morphism; the corresponding cases are omitted from figure 6.8.

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 $\langle t;o \rangle$ is translated into a record construction. The resulting record has one field for each exported name $X \in \text{dom}(o)$, and this field is associated to $o(X)$ where all input parameters on which $X$ depends are $\lambda$-abstracted. Some notation is required here. We write $D^{-1}(X)$ for the list of immediate predecessors of node $X$ in the dependency graph $D$, 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 $(X_1, \ldots, X_n) = D^{-1}(X)$ is such a list, we write $t^{-1}(D^{-1}(X))$ for the list $(x_1, \ldots, x_n)$ of variables associated to the names $(X_1, \ldots, X_n)$ by the input mapping $t$. Finally, we write $\lambda_t(x_1, \ldots, x_n).M$ as shorthand for $\lambda x_1 \ldots \lambda x_n.M$. With all this notation, the field $X$ in the record translating $(t;o)$ is bound to $\lambda_t(D^{-1}(X)).[\text{dom}(X) : O(X)]$.

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 $M \overline{x_1} \ldots \overline{x_n}$.

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 $Y_1, \ldots, Y_p$, 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 $(\{Y_1 \cup \{X_j\}) \setminus \{X\}$. 133
\[ [e : M'] . X : M ] = [e : M'] . X \]
\[ [(i ; o) : \{ I ; O ; D \}] = \]
\[ \{ X = \lambda _ { i , o } . ( D ^ { - 1 } ( X ) ) ; [ o ( X ) : O ( X ) ] \mid X \in \text{dom}(O) \} \]
\[ [(E _ { 1 } : \{ I _ { 1 } ; O _ { 1 } ; D _ { 1 } \}) + (E _ { 2 } : \{ I _ { 2 } ; O _ { 2 } ; D _ { 2 } \}) : \{ I ; O ; D \}] = \]
   let \( e _ { 1 } = [E _ { 1 } : \{ I _ { 1 } ; O _ { 1 } ; D _ { 1 } \}] \) in
   let \( e _ { 2 } = [E _ { 2 } : \{ I _ { 2 } ; O _ { 2 } ; D _ { 2 } \}] \) in
   \( \langle X = e _ { 1 } . X \mid X \in \text{dom}(O _ { 1 }) \rangle ; \)
\[
   Y = e _ { 2 } . Y \mid Y \in \text{dom}(O _ { 2 }) \]
\[ [(E : \{ I ' ; O ' ; D ' \}) \setminus X : \{ I ; O ; D \}] = \]
   let \( e = [E : \{ I ' ; O ' ; D ' \}] \) in
   \( \langle Y = e . Y \mid Y \in \text{dom}(O) \rangle \)
\[ [(E : \{ I ' ; O ' ; D ' \}) \setminus X : \{ I ; O ; D \}] = \]
   let \( e = [E : \{ I ' ; O ' ; D ' \}] \) in
   \( \langle Y = e . Y \mid Y \in \text{dom}(O) \rangle \)
\[ [\text{close } E : \{ I ' ; O ' ; D ' \} : \emptyset ; O ; \emptyset ] = \]
   let \( e = [E : \{ I ' ; O ' ; D ' \}] \) in
   let \( \text{rec } \bar{X} = e . X \ D ^ { - 1 } ( \bar{X} ) \) in
   \( \langle X = \bar{X} \mid X \in \text{dom}(O) \rangle \)
\[ \text{where } \{ \{ X _ { 1 } , \ldots , X _ { n _ { 1 } } \}, \ldots , \{ X _ { p _ { 1 } } , \ldots , X _ { n _ { p } } \} \} \text{ is a serialization of } \text{dom}(O) \text{ against } D ' \]

Figure 6.8: The translation scheme

Thus, we \( \lambda \)-abstract on the corresponding variables, then compute \( X \) by applying \( [E] . X \) to the parameters \( \bar{X} \). Since \( X \) can depend on itself, this application must be done in a \( \text{let rec} \) binding over \( \bar{X} \). Then, we apply \( [E] . Y \) to the parameters that it expects, namely \( \bar{Y} \), which include \( X \).

The only operator that remains to be explained is \( \text{close } E \). Here, we take advantage of the fact that \( \text{close} \) removes all input dependencies to generate code that is more efficient than a sequence of \( \text{freeze} \) operations. We first \( \text{serialize} \) the set of names exported by \( E \) against its dependency graph \( D \). That is, we identify strongly connected components of \( D \), then sort them in topological order. The result is an enumeration \( \{ \{ X _ { 1 } , \ldots , X _ { n _ { 1 } } \}, \ldots , \{ X _ { p _ { 1 } } , \ldots , X _ { n _ { p } } \} \} \) of the exported names where each cluster \( \{ X _ { 1 } , \ldots , X _ { n _ { 1 } } \} \) 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 \( \text{let rec} \) 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.
\[
\begin{align*}
\gamma(x) &= 0 \\
\Gamma \vdash x : \Gamma(x) / \gamma & \quad \text{(VAR)} \\
\Gamma \vdash c : TC(c) / \gamma & \quad \text{(CONST)} \\
\Gamma + \{x : \tau'\} \vdash M : \tau / (\gamma - 1)[x \mapsto d] & \quad \text{(ABSTR)} \\
\Gamma \vdash \lambda x.M : \tau' \rightarrow \tau / \gamma \\
\Gamma \vdash M_1 : \tau' \rightarrow \gamma_1 & \quad \Gamma \vdash M_2 : \tau' / \gamma_2 & \quad \text{(APP)} \\
\Gamma \vdash M : \tau' \rightarrow \tau / \gamma & \quad \Gamma(x) = \tau' \\
\Gamma \vdash M : \tau' / \gamma & \quad \Gamma + \{x : \tau'\} \vdash N : \tau / \gamma[x \mapsto d] & \quad \text{(LET)} \\
\Gamma \vdash \text{let rec } \ldots x_i = M_i \ldots \text{ in } M : \tau / \gamma & \quad \text{(REC)} \\
\forall i : \Gamma \vdash M_i : \tau_i / \gamma & \quad \Gamma \vdash (\ldots X_i = M_i \ldots) : (\ldots X_i : \tau_i \ldots) / \gamma & \quad \text{(RECORD)} \\
\Gamma \vdash M : (\ldots X_j : \tau_j \ldots) / \gamma & \quad 1 \leq i \leq n & \quad \text{(SCL)} \\
\end{align*}
\]

Figure 6.9: Typing rules for $\lambda_c$

### 6.3 Type soundness of the translation

#### 6.3.1 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 ...` In $\lambda_c$, these definitions can either evaluate to a fixpoint (i.e. $M = \lambda x. \lambda y. y$), or get stuck (i.e. $M = \lambda x. x + 1$). In preparation for showing that no term generated by the translation can get stuck, we now equip $\lambda_c$ with a sound type system that guarantees that all recursive definitions are correct. Boudol [13] gave such a type system, however it does not type-check curried function applications with sufficient precision for our purposes. Hence we now define a refinement of Boudol’s type system.

The type system for $\lambda_c$ is defined in figure 6.9. Types, written $\tau$, have the following syntax:

\[
\begin{align*}
\lambda_c \text{ types: } \tau ::= & \quad \text{int} \mid \text{bool} & \quad \text{base types} \\
& \mid \tau_1 \rightarrow \tau_2 & \quad \text{annotated function types} \\
& \mid \{\ldots X_i : \tau_i \ldots\} & \quad \text{record types}
\end{align*}
\]

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 $\text{int} \rightarrow \text{int}$, because the value of $x$ is immediately
needed after application, whereas \( \lambda y z. x + 1 \) has type \( \text{int} \rightarrow \ldots \), 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 \( \infty \), meaning that the variable is not used. Similarly, the typing judgment is of the form \( \Gamma \vdash M : \tau / \gamma \), where \( \gamma \) is a (total) mapping from variables to degrees, indicating how \( M \) uses each variable: \( \gamma(x) = \infty \) means that \( x \) is not free in \( M \); \( \gamma(x) = 0 \) means that the value of \( x \) is needed to evaluate \( M \); and \( \gamma(x) = n + 1 \) means that the value of \( x \) is needed only after \( n + 1 \) function applications, e.g. \( x \) occurs in \( M \) under at least \( n + 1 \) function abstractions.

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

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

\[
d @ 0 = 0 \quad d @ \infty = \infty \quad d @ (n + 1) = d
\]

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 how many times the function \( M_1 \) is going to apply its argument inside its body. However, we know that it will not do so before \( d \) more applications of \( M_1 \) \( M_2 \). Hence, we can take \( d \) for the degree of \( x \) in \( M_1 \) \( M_2 \). Finally, the contributions from the function part \( (\gamma_1 - 1) \) and the argument part \( (d @ \gamma_2) \) are combined with the \( \land \) 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 \( (\gamma(x) - 1) \land d \), 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 \( x_1 \), making \( M_1 \) depend on \( x_1 \) as well. We account for these indirect dependencies via the triangular inequality \( d_{ik} \leq d_{ij} \land d_{jk} \). Finally, the dependencies of the whole let rec are obtained by combining those of its body \( M \) with those arising from the uses of the \( x_i \) in \( M \), either direct \( (d_i @ \gamma_i) \) 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 inequality.

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

### 6.3.2 Soundness of the target language

To simplify the proofs, we prove the soundness on a subset \( \lambda_\subset \) of \( \lambda_c \) that excludes constants, record construction and access, and the let binding. It is entirely straightforward to extend the proofs to the omitted constructs.

#### Properties of degrees

We start the proof with a number of algebraic lemmas on degrees and degree operations. Figure 6.10 re-states the definitions of the operations on degrees. The following lemmas should be read as
Figure 6.10: Summary of degree operations

universally quantified over the degrees $d$, $d'$, $d_1$, $d_2$, $d_3$. We adopt the convention that $@$ has highest precedence, followed by $\land$, and then $+$ and $-.$

**Lemma 18**

1. $(d_1 + 1) @ d_2 \leq d_1 @ d_2 + 1$.
2. $(d_1 \land d_2) @ d_3 = d_1 @ d_3 \land d_2 @ d_3$.
3. $d_1 @ (d_2 \land d_3) = d_1 @ d_2 \land d_1 @ d_3$.
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 $d_1 = 0$, we obtain $0$ on both sides of the equality. If $d_3 = \infty$, both sides are equal to $\infty$. Otherwise we get $d_1 \land d_2$ on both sides.
3. If $d_2 = 0$, both sides are equal to $0$. If $d_2 = \infty$, then $d_2 \land d_3 = d_3$ 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, $d_2 \land d_3 = n \neq 0$, so $d_1 @ (d_2 \land d_3) = d_1 @ d_1 \land d_1 = d_1 @ d_2 \land d_1 @ d_3$.
4. If $d_3 = 0$, both sides are equal to $0$. If $d_3 = \infty$, we obtain $\infty$ on both sides. Otherwise, both sides are equal to $d_1 @ d_2$.
5. Both sides reduce to $\infty$ 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 @.


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, \( d_1 \oplus d_3 = d_1 \) and \( d_1 \oplus d_2 \oplus d_3 = d_1 \oplus d_2 \), hence it simply remains to prove that \( d_1 \leq d_1 \oplus d_2 \). Since \( d_2 \geq 1 \), we have only two cases: either \( d_2 = \infty \), in which case \( d_1 \oplus d_2 = \infty \) which cannot be less than \( d_1 \); or \( d_2 = m + 1 \), in which case \( d_1 \oplus d_2 = d_1 \), and the result holds.

\[ \square \]

**Lemma 19** If \( \gamma \leq (\gamma_1 - 1) \land d \land \gamma_2 \), then there exists \( \gamma_1' \) and \( \gamma_2' \) such that \( \gamma = (\gamma_1' - 1) \land d \land \gamma_2' \) and \( \gamma_1' \leq \gamma \) and \( \gamma_2' \leq \gamma_2 \).

**Proof** We define \( \gamma_1' \) and \( \gamma_2' \) 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 \( d' = (d_1' - 1) \land d \land d_2' \) and \( d_1' \leq d_1 \land d_2' \leq d_2 \).

- 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 \( \infty \) is greater than \( d' \).
- If \( d = n + 1 \), let \( d_1' = n + 2 \) and \( d_2' = d_2 \). By hypothesis we know that \( d' \leq d \land d_2 \). Since \( d_1' - 1 = n + 1 = d' \), we have \( (d_1' - 1) \land d \land d_2' = d_1' - 1 = d' \). 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 18. Finally, \( d_2' \leq d_2 \) trivially holds.

\[ \square \]

**Lemma 20** If \( \gamma \leq (\gamma_1 - 1) \land (x \rightarrow d) \), then there exists \( \gamma_1' \) such that \( \gamma_1' \leq \gamma_1 \) and \( \gamma = (\gamma_1' - 1) \land (x \rightarrow d) \).

**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 \( d' = (d_1' - 1) \land ((x \rightarrow d)(y)) \).

- If \( d_1 = 0 \), then \( d_1' = 0 \) works.
- Otherwise, we take \( d_1' = d_1 + 1 \). This \( d_1' \) is suitable because:
  - Since \( d' \leq d_1 - 1 \), we have \( d_1 + 1 \leq d_1 - 1 + 1 \) and \( d_1 \neq 0 \). By lemma 18, 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 \( d \leq (d_1 - 1) \land (x \rightarrow d)(y) \leq (x \rightarrow d)(y) \) it follows that \( d' \leq (d_1' - 1) \land ((x \rightarrow d)(y)) \).
  - Since \( d_1' - 1 = d' \), we have that \( (d_1' - 1) \land ((x \rightarrow d)(y)) \leq d' \).

\[ \square \]

**Lemma 21** Let \( n \in \mathbb{N} \). If

\[
\gamma' \leq \gamma_0 \land \bigwedge_{i,j \in \{1 \ldots n\}} d_i \oplus d_j \oplus g_j \land \bigwedge_{i \in \{1 \ldots n\}} d_i \oplus g_i
\]

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

\[
\gamma' = \gamma_0' \land \bigwedge_{i,j \in \{1 \ldots n\}} d_i \oplus d_j \oplus \gamma_j' \land \bigwedge_{i \in \{1 \ldots n\}} d_i \oplus \gamma_i'\]
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

$$
\gamma'_{0} \land \bigwedge_{i,j \in \{1 \ldots n\}} d_{i} \circ d_{ij} \circ \gamma'_{j} \land \bigwedge_{i \in \{1 \ldots n\}} d_{i} \circ \gamma'_{i} \leq \gamma'
$$

by definition of $\gamma'$. Moreover, by hypothesis, we know that

$$
\bigwedge_{i,j \in \{1 \ldots n\}} d_{i} \circ d_{ij} \circ \gamma_{j} \geq \gamma' \quad \text{and} \quad \bigwedge_{i \in \{1 \ldots n\}} d_{i} \circ \gamma_{i} \geq \gamma'
$$

hence

$$
\gamma' \leq \gamma'_{0} \land \bigwedge_{i,j \in \{1 \ldots n\}} d_{i} \circ d_{ij} \circ \gamma_{j} \land \bigwedge_{i \in \{1 \ldots n\}} d_{i} \circ \gamma_{i}
$$

and the expected equality follows. □

Lemma 22 If $\gamma[x \rightarrow d] = (\gamma_{1} - 1) \land d_{0} \circ \gamma_2$ then there exist $\gamma'_{1}$, $\gamma'_{2}$, $d_{1}$, $d_{2}$ such that $\gamma_{1} = \gamma'_{1}[x \rightarrow d_{1}]$, $\gamma_{2} = \gamma'_{2}[x \rightarrow d_{2}]$, and $\gamma = (\gamma'_{1} - 1) \land d_{0} \circ \gamma'_{2}$.

Proof Let $d_{1} = \gamma_{1}(x)$ and $d_{2} = \gamma_{2}(x)$. Let $\gamma'_{1}$ be the function associating $\gamma_{1}(y)$ to every variable $y \neq x$ and such that $\gamma'_{1}(x) = \gamma_{1}(x) + 1$, which we can write $\gamma_{1}[x \rightarrow \gamma_{1}(x) + 1]$. Let $\gamma'_{2}$ be the function associating $\gamma_{2}(y)$ to every variable $y \neq x$ and such that $\gamma'_{2}(x) = \infty$, which we can write $\gamma_{2}[x \rightarrow \infty]$. We have trivially $\gamma_{1} = \gamma'_{1}[x \rightarrow d_{1}]$ and $\gamma_{2} = \gamma'_{2}[x \rightarrow d_{2}]$. We now check the third property. On $x$,

$$
\gamma(x) = (\gamma(x) + 1 - 1) \land d_{0} \circ \infty = (\gamma'_{1}(x) - 1) \land d_{0} \circ \gamma'_{2}(x)
$$

On $y \neq x$,

$$
\gamma(y) = (\gamma(y) - 1) \land d_{0} \circ \gamma_2(y) = (\gamma'_{1}(y) - 1) \land d_{0} \circ \gamma'_{2}(y)
$$

□

Lemma 23 If $\gamma[x \rightarrow d] = \gamma_{0} \land \bigwedge_{i,j \in \{1 \ldots n\}} d_{i} \circ d_{ij} \circ \gamma_{j} \land \bigwedge_{i} d_{i} \circ \gamma_{i}$ then there exist $\gamma'_{0}$ and $\gamma'_{i}$ for each $i$, such that $\gamma'_{0}[x \rightarrow d_{0}] = \gamma_{0}$, $\gamma'_{i}[x \rightarrow d_{i}] = \gamma_{i}$, and $\gamma = \gamma'_{0} \land \bigwedge_{i,j \in \{1 \ldots n\}} d_{i} \circ d_{ij} \circ \gamma'_{j} \land \bigwedge_{i} d_{i} \circ \gamma'_{i}$, with $d_{0} = \gamma_{0}(x)$ and $d_{i} = \gamma_i(x)$ for all $i$.

Proof Take $\gamma'_{0} = \gamma_{0}[x \rightarrow \gamma_{x}]$ and $\gamma'_{i} = \gamma_{i}[x \rightarrow \infty]$ for all $i$. We check that the expected properties hold as in the previous proof. □

Weakening lemmas

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

Lemma 24 (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.

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 $\Gamma \vdash \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 / (\gamma - 1)[x \mapsto d]$ with $\tau = \tau_1 \downarrow\uparrow_2 \tau_2$. But it is easy to notice that $(\gamma' - 1)[x \mapsto d] \leq (\gamma - 1)[x \mapsto d]$, so by induction hypothesis, we have a derivation of $\Gamma \vdash \{x \mapsto \tau_1\} \vdash M_1 : \tau_2 / (\gamma' - 1)[x \mapsto d]$. 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 $\Gamma \vdash M_1 : \tau' \downarrow\uparrow \tau / \gamma_1$ and $\Gamma \vdash M_2 : \tau'' / \gamma_2$, with $\gamma = (\gamma_1 - 1) \land d \land \gamma_2$. By lemma 19, we construct $\gamma_1'$ and $\gamma_2'$, such that $\gamma_1' \leq \gamma_1$, $\gamma_2' \leq \gamma_2$ and $\gamma' = (\gamma_1 - 1) \land d \land \gamma_2'$. Applying the induction hypothesis twice, we obtain derivations for $\Gamma \vdash M_1 : \tau' \downarrow\uparrow \tau / \gamma_1'$ 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 $\Gamma \vdash M_1 : \tau' \downarrow\uparrow \tau / \gamma_1$ with $\Gamma(x) = \tau'$ and $\gamma = (\gamma_1 - 1) \land d$. Hence, $\gamma' \leq (\gamma_1 - 1) \land (x \mapsto d)$. Applying lemma 20, we obtain $\gamma_1'$ such that $\gamma_1' \leq \gamma_1$ and $\gamma' = (\gamma_1' - 1) \land (x \mapsto d)$. We can apply rule (appvar) again to derive the expected judgment.

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

\[
\begin{align*}
\Gamma + \{x_j : \tau_j \ldots\} &\vdash N : \tau / \gamma_0[\ldots x_j \mapsto d_j \ldots] \\
\Gamma + \{x_j : \tau_j \ldots\} &\vdash M_i : \tau_i / \gamma_i[\ldots x_j \mapsto d_{ij} \ldots]
\end{align*}
\]

for all $i, j, d_{ij} \geq 1$

\[
\begin{align*}
\gamma = \gamma_0 &\land \left(\bigwedge_{i} \left(\bigwedge_{j} d_{ij} \land d_{ij}\right)\right)
\end{align*}
\]

Using lemma 21, we take $\gamma_N = \gamma'$ and for all $i$, $\gamma_i = \gamma_i$, knowing that $\gamma_N \leq \gamma_0$ and $\gamma' = \gamma_N \land \left(\bigwedge_{i} \left(\bigwedge_{j} d_{ij} \land d_{ij}\right)\right)$. By induction hypothesis, we know how to derive $\Gamma + \{x_j : \tau_j \ldots\} \vdash N : \tau / \gamma_N[\ldots x_j \mapsto d_j \ldots]$. Hence we can derive $\Gamma \vdash M : \tau / \gamma'$. $\square$

**Lemma 25 (degree weakening)** If $\Gamma \vdash M : \tau / \gamma[x \mapsto d]$ and $x \not\in \text{FV}(M)$, 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 $x \not\in \text{FV}(M)$, $x \neq y$. 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: $\Gamma + \{y : \tau_1\} \vdash M_1 : \tau_2 / (\gamma_1 - 1)[y \mapsto d_0]$ and $\tau = \tau_1 \downarrow\uparrow_2 \tau_2$. But, obviously $\gamma = (\gamma_1 - 1)[y \mapsto d_0] = (\gamma_1 - 1)[y \mapsto d_0][x \mapsto d - 1]$. Hence, by induction hypothesis we obtain $\Gamma + \{y : \tau_1\} \vdash M_1 : \tau_2 / (\gamma_1 - 1)[y \mapsto d_0]$ and the expected result follows by rule (abstr).

**Rule (app)**, $M = M_1 \ M_2$. We have $\Gamma \vdash M_1 : \tau' \downarrow\uparrow \tau / \gamma_1$ and $\Gamma \vdash M_2 : \tau'' / \gamma_2$ with $\gamma = (\gamma_1 - 1) \land d \land \gamma_2$. Applying lemma 22, we obtain $d_1, d_2, \gamma_1'$ and $\gamma_2'$ such that $\gamma = (\gamma_1 - 1) \land d \land \gamma_2, \gamma_1'[x \mapsto d_1] = \gamma_1$ and $\gamma_2'[x \mapsto d_2] = \gamma_2$. By induction hypothesis we can derive $\Gamma \vdash M_1 : \tau' \downarrow\uparrow \tau / \gamma_1'$ 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 $x \not\in \text{FV}(M)$. We have a derivation of $\Gamma \vdash M_1 : \tau_1 \downarrow\uparrow \tau_2 / \gamma$ with $\gamma = (\gamma_1 - 1) \land d \land \gamma_2$. Taking $\gamma_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. 

140
We have $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$ and $\gamma = (\gamma'_1 - 1) \land (y \mapsto d_0)$. The first equality is straightforward, and the second equality follows from the facts that $\gamma(x) = \gamma(x) + 1 - 1$, and for any $z \neq x$, $(\gamma_1 - 1) \land (y \mapsto d_0)(z) = ((\gamma'_1 - 1) \land (y \mapsto d_0))(z)$. We then conclude by induction hypothesis as above.

**Rule (rec), $M = \text{let rec } \ldots \ x_i = M_i \ldots \text{ in } N$.** We have

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

and for all $i$

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

with for all $i, j, k, d_{ik} \leq d_{ij} \land d_{jk}$ and for all $i, j, d_{ij} \geq 1$ and $\gamma[x \mapsto d] = \gamma_N \land \bigwedge_i (d_i \land \gamma_i) \land \bigwedge_{i,j} (d_i \land d_{ij} \land \gamma_j)$. Lemma 23 shows the existence of $\gamma_N$ and $\gamma_i$ 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 $\gamma = \gamma'_N \land \bigwedge_i (d_i \land \gamma'_i) \land \bigwedge_{i,j} (d_i \land d_{ij} \land \gamma'_j)$, 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). □

**Lemma 26 (type weakening)** If $\Gamma + \{ x : \tau' \} \vdash M : \tau \ / \ \gamma$ and $x \notin \text{FV}(M)$, then $\Gamma \vdash M : \tau \ / \ \gamma$.

**Proof** Straightforward by induction on the typing derivation. □

**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 $\land$ in relation with what really happens during the reduction of an application.

**Lemma 29 (substitution)** If $\Gamma + \{ x : \tau' \} \vdash M_1 : \tau \ / \ \gamma_1[x \mapsto d]$, and $\Gamma \vdash M_2 : \tau' \ / \ \gamma_2$, with $x \notin \text{FV}(M_2) \cup \text{dom}(\gamma_2)$, then $\Gamma \vdash M_1[x]M_2 : \tau \ / \ \gamma_1 \land d \land \gamma_2$.

**Proof** We proceed by induction on the typing derivation of $M_1$ and case analysis on the last typing rule used. We write $M = M_1[x]M_2$, $\Gamma' = \Gamma + \{ x : \tau' \}$, and $\gamma_0 = \gamma_1 \land d \land \gamma_2$.

**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 $\Gamma \vdash M : \tau \ / \ \gamma_2$. So by lemma 24, it is enough that $\gamma_0 \leq \gamma_2$ or $\gamma_1 \land 0 \land \gamma_2 \leq \gamma_2$, which is true by lemma 18.

If $y \neq x$, then $x \notin \text{FV}(M)$ and $\Gamma + \{ x : \tau' \} \vdash M : \tau \ / \ \gamma_1[x \mapsto d]$, so by lemmas 25 and 26, $\Gamma \vdash M : \tau \ / \ \gamma_1$, and it suffices 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 : \tau_1 \} \vdash M_3 : \tau_2 \ / \ \gamma_3[y \mapsto d]$
with \( \tau = \tau_1 \xrightarrow{d_1} \tau_2 \) and \( \gamma_3 [y \mapsto d_0] = (\gamma_3 [x \mapsto d] - 1)[y \mapsto d_0] = (\gamma_3 - 1)[x \mapsto (d-1); y \mapsto d_0] \). Take \( M'_3 = M_3 \{ x \} M_2 \). By induction hypothesis, we have \( \Gamma + \{ y \mapsto \tau_1 \} \vdash M'_3 : \tau_2 \vdash (\gamma_1 - 1)[y \mapsto d_0] \wedge \gamma_2 \). Since \( y \) is fresh, it does not occur in \( \gamma_2 \), hence

\[
\begin{align*}
(\gamma_1 - 1)[y \mapsto d_0] \wedge (d - 1) @ \gamma_2 \\
= ((\gamma_1 - 1) \wedge (d - 1) @ \gamma_2)[y \mapsto d_0] \\
= ((\gamma_1 - 1) \wedge (d @ \gamma_2 - 1))[y \mapsto d_0] \text{ by lemma 18} \\
= ((\gamma_1 \wedge d @ \gamma_2 - 1)\}y \mapsto d_0] = (\gamma_0 - 1)[y \mapsto d_0]
\end{align*}
\]

Hence, rule (abstr) concludes \( \Gamma \vdash \lambda y M'_3 : \tau_1 \xrightarrow{d_3} \tau_2 / \gamma_0 \), which is the expected result.

**Rule (app),** \( M_1 = M_3 M_4 \). We have \( \Gamma' \vdash M_3 : \tau'' \xrightarrow{d_3} \tau / \gamma_3 \) and \( \Gamma' \vdash M_4 : \tau'' / \gamma_4 \) and \( \gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge d_0 @ \gamma_4 \). By lemma 22, if \( d_3 = \gamma_3(x) \) and \( d_4 = \gamma_4(x) \), there exists \( \gamma'_3 \) and \( \gamma'_4 \) such that \( \gamma'_3[x \mapsto d_3] = \gamma_3 \), \( \gamma'_4[x \mapsto d_4] = \gamma_4 \), and \( \gamma_1 = (\gamma'_3 - 1) \wedge d_0 @ \gamma'_4 \). By induction hypothesis, if \( M'_3 = M_3 \{ x \} M_2 \) and \( M'_4 = M_4 \{ x \} M_2 \), then \( \Gamma \vdash M'_3 : \tau'' \xrightarrow{d_3} \tau / \gamma'_3 \wedge d_3 @ \gamma_2 \) and \( \Gamma \vdash M'_4 : \tau'' / \gamma'_4 \wedge d_4 @ \gamma_2 \), so by rule (app)

\[
\Gamma \vdash M : \tau / ((\gamma'_3 \wedge d_3 @ \gamma_2 - 1) \wedge d_0 @ (\gamma'_4 \wedge d_4 @ \gamma_2))
\]

Moreover, by lemma 18, the degree environment is equal to

\[
(\gamma'_3 - 1) \wedge (d_3 @ \gamma_2 - 1) \wedge (d_0 @ d_4 @ \gamma_2) \\
= \gamma_1 \wedge (d_3 @ \gamma_2 - 1) \wedge (d_0 @ d_4 @ \gamma_2) \\
= \gamma_1 \wedge ((d_3 - 1) \wedge d_0 @ d_4) @ \gamma_2 \\
= \gamma_1 \wedge d @ \gamma_2 \\
= \gamma_0
\]

**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** \( y = x \). Then, \( M = M'_3 M_2 \), where \( M'_3 = M_3 \{ x \} M_2 \). The typing hypothesis implies \( \Gamma' \vdash M_3 : \tau'' \xrightarrow{d_3} \tau / \gamma_3 \) \( \tau \) and \( \Gamma'(y) = \Gamma'(x) = \tau' = \tau'' \) and \( \gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0) \). Take \( \gamma'_3 = \gamma_3[x \mapsto \gamma_3(x) + 1] \). 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

\[
\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_3} \tau / \gamma'_3 [x \mapsto \gamma_3(x)]
\]

Hence, by induction hypothesis we have

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

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

\[
\Gamma \vdash M : \tau / ((\gamma'_3 \wedge d_3 @ \gamma_2 - 1) \wedge d_0 @ \gamma_2)
\]

But \( \gamma_0 = (\gamma'_3 - 1) \wedge d @ \gamma_2 \). Since \( d = (d_3 - 1) \wedge d_0 \), it follows that

\[
\gamma_0 = (\gamma'_3 - 1) \wedge (d_3 @ \gamma_2 - 1) \wedge d_0 @ \gamma_2
\]

Hence, we have derived the desired judgment.

**Case** \( y \neq x \). Then, \( M = M'_3 y \), where \( M'_3 = M_3 \{ x \} M_2 \). By typing hypothesis, we have \( \Gamma' \vdash M_3 : \tau'' \xrightarrow{d_3} \tau / \gamma_3 \) \( \tau \) 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 \( \gamma_1 = (\gamma'_3 - 1) \land (y \mapsto d_0), \) and \( \gamma'_3[x \mapsto \gamma_3(x)] = \gamma_3. \) Thus we rewrite the premise \((\ast)\) as follows:

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

By induction hypothesis, it follows that

\[ \Gamma \vdash M'_3 : \tau'' \xrightarrow{d_3} \tau / \gamma'_3 \land d_3 @ \gamma_2 \]

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

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

which yields by lemma 18

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

Moreover,

\[
\begin{align*}
\gamma_0 &= \gamma_1 \land d @ \gamma_2 \\
&= (\gamma'_3 - 1) \land (y \mapsto d_0) \land d @ \gamma_2 \\
&= (\gamma'_3 - 1) \land (y \mapsto d_0) \land (d_3 - 1) @ \gamma_2
\end{align*}
\]

(because \( \gamma_1[x \mapsto d] = (\gamma_3 - 1) \land (y \mapsto d_0) \))

\[ = (\gamma'_3 - 1) \land (y \mapsto d_0) \land (d_3 @ \gamma_2 - 1) \quad \text{(by lemma 18)} \]

Thus, the expected result holds.

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

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

We write \( N' = N \{x\} M_2 \) and for all \( i, N'_i = N_i \{x\} M_2 \). We have \( \gamma_1[x \mapsto d] = \gamma_N \land (\bigwedge_i d_i @ \delta_i) \land (\bigwedge_{i,j} d_{ij} @ \delta_{ij}) \). Lemma 23 shows that we can construct \( \gamma'_N \) and \( \delta'_i \) for all \( i \) such that \( \gamma'_N[x \mapsto d_N] = \gamma_N \), and \( \delta'_i[x \mapsto d'_i] = \delta_i \) for all \( i \) and \( \gamma_1 = \gamma'_N \land (\bigwedge_i d_i @ \delta_i') \land (\bigwedge_{i,j} d_{ij} @ \delta'_{ij}) \), with \( d_N = \gamma_N(x) \) and \( d'_i = \delta_i(x) \) for each \( i \). Thus, the two premises can be rewritten as follows:

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

By induction hypothesis, it follows that

\[ \Gamma + \{ \ldots \ x_j : \tau_j \ldots \} \vdash N' : \tau / \gamma'_N[\ldots \ x_j \mapsto d_j \ldots] \land d_N @ \gamma_2 \]

for all \( i, \Gamma + \{ \ldots \ x_j : \tau_j \ldots \} \vdash N'_i : \tau_i / \delta'_i[\ldots \ x_j \mapsto d_j \ldots] \land d'_i @ \gamma_2 \)

Since the \( x_i \)s are fresh we have \( \gamma'_N[\ldots \ x_j \mapsto d_j \ldots] \land d_N @ \gamma_2 = (\gamma'_N \land d_N @ \gamma_2)[\ldots \ x_j \mapsto d_j \ldots] \) and for all \( i, \delta'_i[\ldots \ x_j \mapsto d_j \ldots] \land d'_i @ \gamma_2 = (\delta'_i \land d'_i @ \gamma_2)[\ldots \ x_j \mapsto d_j \ldots] \). We can therefore apply rule (rec) to obtain

\[
\Gamma \vdash M : \tau / \gamma'_N \land d_N @ \gamma_2 \land \bigwedge_{i,j} d_i @ d_{ij} @ (\delta'_i \land d'_i @ \gamma_2) \land \bigwedge_i d_i @ (\delta'_i \land d'_i @ \gamma_2)
\]

143
According to lemma 18, the degree environment above is equal to
\[
\gamma_N \land (\bigwedge_i (d_i @ \delta_i)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta_j)) \land (\bigwedge_i (d_i @ \delta'_i)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta'_j)) \land (\bigwedge_i (d_i @ d_{ij} @ \gamma_2)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \gamma_2))
\]

To obtain the expected result, it suffices to prove that this degree environment is equal to \(\gamma_0\). Since
\[
\gamma_1[x \mapsto d] = \gamma_N \land (\bigwedge_i (d_i @ \delta_i) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta_j))
\]
we know that
\[
d = \gamma_N(x) \land (\bigwedge_i (d_i @ \delta_i(x)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta_j(x)))
\]
Therefore, \(d = d_N \land (\bigwedge_i (d_i @ d_{ij}^c) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ d_{ij}^c))\). It follows that
\[
\gamma_0 = \gamma_1 \land d @ \gamma_2
\]
\[
= \gamma_N \land (\bigwedge_i (d_i @ \delta'_i)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta'_j)) \land (\bigwedge_i (d_i @ \delta_i)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \delta_j)) \land (\bigwedge_i (d_i @ \gamma_2)) \land (\bigwedge_{i,j} (d_i @ d_{ij} @ \gamma_2)) \land (\bigwedge_i (d_i @ d_{ij} @ \gamma_2))
\]
This completes the proof. \(\square\)

We now extend the previous lemma to the case of parallel substitution, exploiting the fact that
\(M \{ \ldots \ x_i \mapsto M_i \ldots \}\) is equal to \(M \{ x_1 \mapsto y_1 \ldots \ x_n \mapsto y_n \} M_1 \ldots \ y_n M_n\), 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 27** If \(\Gamma + \{ x : \tau \} \vdash M : \tau / \gamma[x \mapsto d]\) and \(y \notin \text{FV}(M)\), then \(\Gamma + \{ y : \tau \} \vdash M\{ x \} y : \tau / \gamma[y \mapsto d]\).

**Proof** Easy induction on the typing derivation of \(M\). \(\square\)

**Lemma 31** (parallel substitution) Assume \(\Gamma + \{ \ldots x_i \mapsto \tau_i \ldots \} \vdash M : \tau / \gamma_M[\ldots x_i \mapsto d_i \ldots]\) and for all \(j \in \{1 \ldots n\}\), \(\Gamma \vdash M_j : \tau_j / \gamma_j\) with for all \(i, j\), \(x_i \notin \text{FV}(M_j) \cup \text{dom}(\gamma_j)\). Then, \(\Gamma \vdash M \{ \ldots x_i \mapsto M_i \ldots \} : \tau / \gamma_M \land \bigwedge_i d_i @ \gamma_i\).

**Proof** Write \(M \{ \ldots x_i \mapsto M_i \ldots \}\) as \(M \{ x_1 \} y_1 \ldots \{ x_n \} y_n \{ y_n \} M_1 \ldots \{ y_n \} M_n\) where the \(y_i\) are fresh. We first apply lemma 27 \(n\) times to obtain \(\Gamma + \{ \ldots y_i : \tau_i \ldots \} \vdash M \{ x_1 \} y_1 \ldots \{ x_n \} y_n : \tau / \gamma_M[\ldots y_i \mapsto d_i \ldots]\). Then we apply lemma 29 \(n\) times again, successively using the \(n\) typing hypotheses for the \(M_i\). This leads to the desired judgment. \(\square\)
Substitution by a variable

We now state and prove a stronger variant of lemma 29 for the case where we substitute a variable by another variable. This alternate substitution lemma is distinct from lemma 27: here, y is not supposed to be fresh, and this is why former occurrences of y must be taken into account, which is done through the \( \land \) operation.

**Lemma 30 (substitution by a variable)** If \( \Gamma + \{ x \mapsto \tau' \} \vdash M : \tau / \gamma [x \mapsto d] \) and \( \Gamma(y) = \tau' \), then \( \Gamma \vdash M \{ x \} y : \tau / \gamma \land (y \mapsto d) \).

**Proof** We write \( \Gamma' = \Gamma + \{ x \mapsto \tau' \} \) and \( M' = M \{ x \} y \) 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

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

which yields

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

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

**Rule (app)**, \( M = M_1 \ M_2 \). The typing hypothesis entails \( \Gamma' \vdash M_1 : \tau / \gamma_1 \) and \( \Gamma' \vdash M_2 : \tau' / \gamma_2 \) with \( \gamma [x \mapsto d] = (\gamma_1 - 1) \land d_0 @ \gamma_2 \). 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:

\[
\gamma_1 = \gamma'_1 [x \mapsto \gamma_1 (x)] \quad \gamma_2 = \gamma'_2 [x \mapsto \gamma_2 (x)] \quad \gamma = (\gamma'_1 - 1) \land d_0 @ \gamma'_2
\]

By induction hypothesis, we can derive

\[
\Gamma \vdash M_1 \{ x \} y : \tau'' / \gamma'_2 \land (y \mapsto \gamma_2 (x)) \quad \Gamma \vdash M_2 \{ x \} y : \tau'' / \gamma'_2 \land (y \mapsto \gamma_2 (x))
\]

The degree environment in the conclusion is equal to

\[
(\gamma'_1 - 1) \land d_0 @ \gamma'_2 \land (y \mapsto ((\gamma_1 (x) - 1) \land d_0 @ \gamma_2 (x))) = \gamma \land (y \mapsto d)
\]

The desired result follows.

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

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

\[
\Gamma \vdash M_1 \{ x \} y : \tau ' / \gamma'_1 \land (y \mapsto \gamma_1 (x))
\]
Since $\Gamma(y) = \tau'$, rule (appvar) concludes
\[ \Gamma \vdash M' : \tau / (\gamma'_1 - 1) \land (y \mapsto (\gamma_1(x) - 1)) \land (y \mapsto d_0) \]
but the degree environment in this conclusion is equal to $(\gamma'_1 - 1) \land (y \mapsto ((\gamma_1(x) - 1) \land d_0))$, that is, $\gamma \land (y \mapsto d)$. This is the expected result.

**Case** $z \neq x$. Define $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. We have $\gamma = (\gamma'_1 - 1) \land (z \mapsto d_0)$ and $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$. By induction hypothesis, we obtain
\[ \Gamma \vdash M_i \{x\} y : \tau'' \overset{d_0}{\longrightarrow} \tau / (\gamma'_1 \land (y \mapsto \gamma_1(x))) \]
Since $\Gamma(z) = \tau''$, we derive by rule (appvar)
\[ \Gamma \vdash M' : \tau / (\gamma'_1 - 1) \land (y \mapsto (\gamma_1(x) - 1)) \land (z \mapsto d_0) \]
The latter degree environment is equal to $\gamma \land (y \mapsto (\gamma_1(x) - 1))$, that is, $\gamma \land (y \mapsto d)$, as required to establish the result.

**Rule** (rec), $M = \text{let rec } \ldots x_i = M_i \ldots \text{ in } N$ where the $x_i$ are fresh. The premises of rule (rec) hold:
\[ \Gamma' + \{\ldots x_i :: \tau_i \ldots\} \vdash M_j : \tau_j / (\gamma_j[\ldots x_j \mapsto d_{ji} \ldots] \land (y \mapsto d_{0j})) \text{ for all } j \]
\[ \Gamma' + \{\ldots x_i :: \tau_i \ldots\} \vdash N : \tau / (\gamma_N[\ldots x_i \mapsto d_i \ldots] \land (y \mapsto d)) \text{ for all } i, j, d_{ij} \geq 1 \]
Moreover, $\gamma[x \mapsto d] = \gamma_N \land (\bigwedge_i d_i \land (y \mapsto \gamma_i))$. By lemma 23, we can construct $\gamma'_N$ and $\gamma'_i$ for each $i$ satisfying the following conditions: $\gamma = \gamma'_N \land (\bigwedge_i d_i \land (y \mapsto \gamma'_i))$, $\gamma_N = \gamma'_N[x \mapsto d_N]$, 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:
\[ \Gamma + \{\ldots x_i :: \tau_i \ldots\} \vdash M_j \{x\} y : \tau_j / (\gamma'_j[\ldots x_j \mapsto d_{ji} \ldots] \land (y \mapsto d_{0j})) \text{ for all } j \]
\[ \Gamma + \{\ldots x_i :: \tau_i \ldots\} \vdash N \{x\} y : \tau / (\gamma'_N[\ldots x_i \mapsto d_i \ldots] \land (y \mapsto d_N)) \]
From these premises, rule (rec) derives $\Gamma \vdash M' : \tau / \gamma'$, where
\[ \gamma' = (\bigwedge_i d_i \land (y \mapsto d_N)) \land (\bigwedge_{i,j} d_i \land (y \mapsto \gamma'_j)) \]
\[ \land (\bigwedge_i d_i \land (y \mapsto \gamma'_i)) \]
\[ = \gamma \land (y \mapsto (d_N \land (\bigwedge_{i,j} d_i \land (y \mapsto d'_j) \land (\bigwedge_{i,j} d_i \land (y \mapsto d'_i)))) \land (y \mapsto d)) \]
This concludes the proof. □

**Soundness**

The soundness of $\lambda$’s type system (theorem 3) is, as usual, a corollary of two properties: subject reduction (lemma 32) and progress (lemma 33). We start with a technical lemma on recursive definitions arising from the reduction of a let rec term.
Lemma 28 Assume $\Gamma + \{ \ldots x_i : \tau_i \ldots \} \vdash M_j : \tau_j / \gamma_j [\ldots, x_i \mapsto d_{ij} \ldots]$ for all $j \in \{1 \ldots n\}$. Further assume that for all $i, j$, $d_{ij} \geq 1$ and for all $i, j, k$, $d_{ik} \leq d_{ij} \odot d_{jk}$. Then, for any $i_0 \in \{1 \ldots n\},$

$$\Gamma \vdash \text{rec \ldots } x_i = M_i \ldots \text{ in } M_{i_0} : \tau_{i_0} / \gamma_{i_0} \land \bigwedge_i d_{i_0 i} @ \gamma_i$$

Proof By application of rule (rec), we obtain

$$\Gamma \vdash \text{rec \ldots } x_i = M_i \ldots \text{ in } M_{i_0} : \tau_{i_0} / \gamma_{i_0} \land \bigwedge_i d_{i_0 i} @ \gamma_j \land \bigwedge_i d_{i_0 i} @ \gamma_i$$

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

$$\bigwedge_i d_{i_0 i} @ \gamma_j \land \bigwedge_i d_{i_0 i} @ \gamma_i = \bigwedge_i d_{i_0 i} @ \gamma_i$$

and the expected result follows. □

Lemma 32 (subject reduction) If $\Gamma \vdash M : \tau / \gamma$ and $M \longrightarrow M'$, 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 $M$ can end 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

$$\frac{\Gamma \vdash \{ x \mapsto \tau' \} \vdash M_1 : \tau / (\gamma_0 - 1)[x \mapsto d]}{\Gamma \vdash \lambda x M_1 : \tau' \vdash \tau / \gamma_0} \frac{\Gamma(\gamma) = \tau'}{\Gamma(\gamma) = \tau'}$$

Moreover, $\gamma = (\gamma_0 - 1) \land (y \mapsto d)$ and $M' = M_1 \{x\} y$. By lemma 30, we have

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

which is the expected result.

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

$$\frac{\Gamma \vdash \{ x \mapsto \tau' \} \vdash M_1 : \tau / (\gamma_1 - 1)[x \mapsto d]}{\Gamma \vdash \lambda x M_1 : \tau' \vdash \tau / \gamma_1} \frac{\vdash \Gamma \vdash v : \tau' / \gamma_2}{\vdash \Gamma \vdash v : \tau' / \gamma_2}$$

Moreover, $M' = M_1 \{x\} v$ and $\gamma = (\gamma_1 - 1) \land d \odot \gamma_2$. By lemma 29, it follows that $\Gamma \vdash M' : \tau / \gamma$, as expected.

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

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

for all $i$, $\Gamma + \{ x_j : \tau_j \ldots \} \vdash v_i : \tau_i / \gamma_i \ldots x_j \mapsto d_{ij} \ldots$ for all $i, j, d_{ij} \geq 1$

and $d_{ik} \leq d_{ij} \odot d_{jk}$.
By lemma 28, it follows that

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

By lemma 31, we obtain

$$\Gamma \vdash M' : \tau / \gamma_N \land \left( \bigwedge_i d_i @ (\gamma_i \land \bigwedge_j d_{ij} @ \gamma_j) \right)$$

which is identical to the expected result

$$\Gamma \vdash M' : \tau / \gamma_N \land \left( \bigwedge_i d_i @ \gamma_i \right) \land \left( \bigwedge_{ij} d_{ij} @ \gamma_j \right)$$

**Reduction rule (context)**, $M = \text{E} [M_1]$, $M_1 \rightarrow M'_1$ and $M' = \text{E} [M'_1]$. The result follows by structural induction and case analysis over the context $E$. The only point worth mentioning is that in the case $E = v \Box$ and the typing derivation ends with rule (appvar), then $M_1$ can only be a variable, and therefore cannot reduce. $\Box$

**Lemma 33 (progress)** If $\Gamma \vdash M : \tau / \gamma$ and $\gamma \geq 1$, then either $M$ is a value, or there exists $M'$ such that $M \rightarrow M'$.

**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 $\lambda$-abstraction, i.e. a value.

**Rule (app).** $M = M_1 M_2$. We have $\Gamma \vdash M_1 : \tau' \overset{d}{\rightarrow} \tau / \gamma_1$ and $\Gamma \vdash M_2 : \tau' / \gamma_2$. Moreover, $\gamma = (\gamma_1 - 1) \land d @ \gamma_2$.

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 $\Box M_2$. If $M_1$ is a value and $M_2$ reduces, $M$ also reduces via the context $M_1 \Box$. If both $M_1$ and $M_2$ are values, the type $\tau' \overset{d}{\rightarrow} \tau$ 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 the (beta) rule to reduce $M$.

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

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

**Theorem 3 (soundness of $\lambda$)** 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 6.3.2. 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.
- $D^{-1}(X) = (X_1, \ldots, X_n)$ is the list of the predecessors of $X$ in $D$, ordered lexicographically.

- $D(X, Y) = \min \{ v \mid X \xrightarrow{v} Y \in D \}$ (with the convention that $D(X, Y) = \infty$ if $D$ contains no edges from $X$ to $Y$)

- $FCT_D(X, I) = (M_1, \ldots, M_n)$, for $Pred(D) \subseteq dom(I)$, where
  - $D^{-1}(X) = (X_1, \ldots, X_n)$
  - for all $i \in \{1 \ldots n\}$, $I(X_i) = M_i$ and $D(X_i, X) = v_i$.

- $Pred(D) = \{ X \mid X \xrightarrow{v} Y \in D, X,Y \in Names, v \in Vals \}$

- $Suc(D) = \{ Y \mid X \xrightarrow{v} Y \in D, X,Y \in Names, v \in Vals \}$

\begin{figure}[h]
\centering
\small
\begin{align*}
[τ_1 \rightarrow τ_2] & = \quad τ_1 \xrightarrow{σ} τ_2 \\
[\text{int}] & = \quad \text{int} \\
[\text{bool}] & = \quad \text{bool} \\
[I; O; D] & = \quad \langle X : O(X) \rangle_{X,D,I} \mid X \in \text{dom}(O) \text{ if } \vdash \{ I; O; D \} \\
[M]_{X,D,I} & = \quad [M_1]_{\frac{e_i}{e_i + 1}} [M_2]_{\frac{e_i + 2}{e_i + 1}} \cdots [M_n]_{\frac{e_n}{e_n + 1}} [M] \\
& \quad \text{where } (M_1, \ldots, M_n) = FCT_D(X, I)
\end{align*}
\caption{Operations on graphs}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\tau_1 & = \tau_2 \\
\text{int} & = \text{int} \\
\text{bool} & = \text{bool} \\
\{ I; O; D \} & = \langle X : O(X) \rangle_{X,D,I} \mid X \in \text{dom}(O) \text{ if } \vdash \{ I; O; D \} \\
[M]_{X,D,I} & = [M_1]_{\frac{e_i}{e_i + 1}} [M_2]_{\frac{e_i + 2}{e_i + 1}} \cdots [M_n]_{\frac{e_n}{e_n + 1}} [M] \\
& \quad \text{where } (M_1, \ldots, M_n) = FCT_D(X, I)
\end{align*}
\caption{Translation of types}
\end{figure}

**Lemma 29 (substitution)** If $Γ + \{ x \rightarrow τ' \} \vdash M : τ / γ_1 [x \rightarrow d]$, and $Γ, x \notin FV(M_2) \cup \text{dom}(γ_2)$, then $Γ \vdash M_1 \{ x \rightarrow M_2 \} : τ / γ_1 \land d @ γ_2$.

**Lemma 30 (substitution by a variable)** If $Γ + \{ x \rightarrow τ' \} \vdash M : τ / γ[x \rightarrow d]$ and $Γ(y) = τ'$, then $Γ \vdash M \{ x \rightarrow y \} : τ / γ \land (y \rightarrow d)$.

**Lemma 31 (parallel substitution)** If $Γ + \{ \ldots x_i \rightarrow \tau_i \ldots \} \vdash M : τ / γ_M[\ldots x_i \rightarrow d_i \ldots]$, and for all $j \in \{1 \ldots n\}$, $Γ \vdash M_j : τ_j / γ_j$ with for all $i,j$, $x_i \notin FV(M_j) \cup \text{dom}(γ_j)$, then $Γ \vdash M[\ldots x_i \rightarrow M_i \ldots] : τ / γ_M \land \bigwedge_i d_i @ γ_i$.

The soundness of $λ_c$ then follows from the well-known properties of subject reduction (reduction preserves typing) and progress (well-typed terms are not stuck).

**Lemma 32 (subject reduction)** If $Γ \vdash M : τ / γ$ and $M \rightarrow M'$, then $Γ \vdash M : τ / γ$.

**Lemma 33 (progress)** If $Γ \vdash M : τ / γ$ and $γ \geq 1$, then either $M$ is a value, or there exists $M'$ such that $M \rightarrow M'$.

\[ \square \]

### 6.3.3 Soundness of the translation

The goal of this section is to prove the soundness of our approach, in the sense that a well-typed $\lambda_b$ expression translates to a well-typed $\lambda_c$ expression. The soundness of $\lambda_c$ then ensures that the translation evaluates correctly.
Core terms:
\[ \mathcal{C} := x^M \mid c^M \] variables, constants
\[ \lambda x \mathcal{C}^M \mid (\mathcal{C}_1 \mathcal{C}_2)^M \] abstraction, application
\[ \overline{E} \cdot X^M \] component projection

Mixin terms:
\[ \mathcal{E} := \mathcal{C} \] core term
\[ (\ell ; \overline{\sigma})^M \] mixin structure
\[ (+ \overline{E}_1 \overline{E}_2)^M \] sum
\[ (\mathcal{E}[X \leftarrow Y])^M \] rename \( X \) to \( Y \)
\[ (\overline{E}' \cdot X)^M \] freeze \( X \)
\[ (\mathcal{E} \setminus X)^M \] delete \( X \)
\[ (\text{close} \overline{E})^M \] close

Output assignments:
\[ \overline{\sigma} := X_i^{i \in I} \overline{E}_i \]

Figure 6.13: Syntax of type-annotated terms

To state the soundness of the translation, we need to set up a translation from source types to \( \lambda_c \) types. We start by defining useful operations on graphs and signatures in figure 6.11. We define \( \text{FCT}_D(X, I) \) as the list of the types and valuations of the predecessors of \( X \) in \( D \) according to \( I \), ordered lexicographically. Then, \( \text{Pred}(D) \) and \( \text{Suc}(D) \) are simply the sets of predecessors and successors of any node in \( D \). The translation of types is presented in figure 6.12. A natural translation for environments follows, defined by \( [\Gamma] = [\Gamma] \circ \Gamma \). Moreover, we define the initial degree environment corresponding to a type environment as \( d^0(\Gamma) = 0 \circ \Gamma \), that is to say the function equal to 0 on \( \text{dom}(\Gamma) \) and \( \infty \) elsewhere. In the sequel, we will often use valuations as degrees. It is worth noticing that for all valuations \( v_1 \) and \( v_2 \), \( \min(v_1, v_2) = v_1 \land v_2 = v_1 \otimes v_2 \).

As the translation operates on annotated well-typed terms, we define an annotated syntax in figure 6.13. 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 now turn to proving theorem 4: the translation of a well-typed source term is a well-typed \( \lambda_c \)-term.

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

**Lemma 34 (single let rec)** The following typing rule is admissible for the type system of \( \lambda_c \).
\[
\Gamma \vdash \{ x \mapsto \tau' \} \vdash M : \tau / \gamma_1 [x \mapsto d] \quad \Gamma \vdash \{ x \mapsto \tau' \} \vdash N : \tau' / \gamma_2 [x \mapsto d'] \quad d' \geq 1
\]
\[
\Gamma \vdash \text{let rec} \ x = N \ in \ M : \tau / \gamma_1 \land d \otimes \gamma_2
\]

**Lemma 35 (n abstractions)** The following typing rule is admissible for the type system of \( \lambda_c \).
\[
\Gamma \vdash \{ \ldots x_i : \tau_i \ldots \} \vdash M : \tau / (\gamma - n)[\ldots x_i \mapsto d_i \ldots]
\]
\[
\Gamma \vdash \lambda (x_1, \ldots, x_n).M : \tau_1 \xrightarrow{d_1 + (n - 1)} \tau_2 \xrightarrow{d_2 + (n - 2)} \ldots \tau_n \xrightarrow{d_n} \tau / \gamma
\]

**Lemma 36 (n applications)** The following typing rule is admissible for the type system of \( \lambda_c \).
\[
\Gamma \vdash M : \tau_1 \xrightarrow{d_1 + (n - 1)} \tau_2 \xrightarrow{d_2 + (n - 2)} \ldots \tau_n \xrightarrow{d_n} \tau / \gamma \quad \Gamma(x_i) = \tau_i \text{ for } i = 1, \ldots, n
\]
\[
\Gamma \vdash M(x_1, \ldots, x_n) : \tau / (\gamma - n) \land (\ldots x_i \mapsto d_i \ldots)
\]

150
We now prove two technical lemmas on the typing of sub-expressions that occur when translating the close and freeze operators.

**Lemma 37 (translation of close)** Assume $\Gamma \vdash e : \mathbb{[}[I; O; D] / d^\varnothing(\Gamma))$. Let $X_1, \dotsc, X_n$ be names such that $\underaccent{a}X \notin dcm(\Gamma)$ and $O(X_i) = I(X_i)$ and $D(X_i, X_j) \neq 0$ for $i, j \in \{1, \dotsc, n\}$. Furthermore assume that for all immediate predecessors $X$ of one of the $X_i$ in $D$, either $X$ is one of the $X_i$, or $\Gamma(\overline{X}) = I(\overline{X})$. Let $M$ be an expression and $\tau$ be a type such that $\Gamma' \vdash M : \tau / d^\varnothing(\Gamma')$, where $\Gamma' = \Gamma + \{X_1 : O(X_1), \dotsc, X_n : O(X_n)\}$. Then,

$$\Gamma \vdash \text{let rec } \overline{X}_1 = e.X_1 \overline{D}^{-1}(X_1) \text{ and } \dotsc \text{ and } \overline{X}_n = e.X_n \overline{D}^{-1}(X_n) \text{ in } M : \tau / d^\varnothing(\Gamma)$$

**Proof** By definition of the translation of a mixin signature, and the hypotheses on $\Gamma$, the conditions of lemma 36 are met, and we obtain

$$\Gamma' \vdash e.X_1 \overline{D}^{-1}(X_1) : O(X_1) / d^\varnothing(\Gamma) \land (\overline{X} \to D(X, X_i) \mid X \in D^{-1}(X_i))$$

Since $\underaccent{a}X \notin dcm(\Gamma)$ for all $j$, the degree environment above is pointwise greater or equal to $d^\varnothing(\Gamma)\overline{X}_j \to D(X_j, X_i) \mid j \in \{1, \dotsc, n\}$. Thus, by lemma 24, it follows that

$$\Gamma' \vdash e.X_1 \overline{D}^{-1}(X_1) : O(X_1) / d^\varnothing(\Gamma) \overline{X}_j \to D(X_j, X_i) \mid j \in \{1, \dotsc, n\}$$

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 24 to its conclusion, we obtain the desired result. □

**Lemma 38 (translation of freeze)** Assume $\Gamma \vdash e : \mathbb{[}[I; O; D] / d^\varnothing(\Gamma)$, where $e$ is a variable distinct from $\overline{X}$ for all names $X$. Let $X$ be a name such that $I(X) = O(X)$. Write $D' = D/X$ and $\Gamma' = I_{\overline{X}}$. Then, for all names $Y \in dcm(O)$, if $X \notin D^{-1}(Y)$ we have

$$\Gamma \vdash e.Y : \mathbb{[}[O(Y)]_{Y, D', \Gamma'} / d^\varnothing(\Gamma)$$

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

$$\Gamma \vdash \text{let rec } \overline{X} = e.X \overline{D}^{-1}(X) \text{ in } e.Y \overline{D}^{-1}(Y') \vdash \mathbb{[}[O(Y)]_{Y, D', \Gamma'} / d^\varnothing(\Gamma)$$

**Proof** Recall the definition of $D'$:

$$D' = D/X = (D \cup D_{\text{around}}) \setminus D_{\text{remove}}$$

where $D_{\text{around}} = \{Z \mid \exists Z' \in D(\overline{Y}) \exists X \in D, (X \to Y) \in D' \}$ and $D_{\text{remove}} = \{X \to Y \mid Y \in \text{Names}, v \in \{0, 1\}\}$.

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

Consider now the case $X \in D^{-1}(Y)$. We have $D^{-1}(Y) = (D^{-1}(Y) \cup D^{-1}(X)) \setminus \{X\}$. Define $\Gamma' = \Gamma + \{Z : [I(Z)] \mid Z \in D^{-1}(Y')\}$. By lemma 36, and using the fact that $e$ is not one of the $Z$, it follows that

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

and

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

Notice that $D(X, X) \geq 1$, because otherwise the graph $D$ would not be safe, making the signature $\{I; O; D\}$ ill-formed. In addition, $O(X) = I(X)$. The conditions of lemma 34 are therefore met, and we obtain $\Gamma' \vdash \text{let rec } \overline{X} = e.X \overline{D}^{-1}(X) \text{ in } e.Y \overline{D}^{-1}(Y') : [O(Y)] / \gamma$ where

$$\gamma = \{e \to 0; \overline{Z} \to D(Z, X) \mid Z \neq X, Z \in D^{-1}(X)\}$$

and

$$\{e \to 0; \overline{Z} \to D(Z, Y) \mid Z \neq X, Z \in D^{-1}(Y)\}$$

151
By definition of $D' = D! X$, $\gamma$ is equal to \{ $e \rightarrow 0; Z \rightarrow D'(Z, Y)$ | $Z \in D^{-1}(Y)$ \}. Applying lemma 35, we obtain

\begin{align*}
\Gamma & \vdash \lambda D'' \overline{\lambda X \cdot Y}.
\end{align*}

\begin{align*}
\text{let rec } \overline{X} = e. X \rightarrow D^{-1}(X) \text{ in e.} \ \overline{Y} \rightarrow D^{-1}(Y): \llbracket O(X) \rrbracket_{X, D, \Gamma} \ / \ \{ e \rightarrow 0 \}
\end{align*}

which implies the desired result by weakening. \hfill \square

**Theorem 4 (soundness of the translation)** If $\Gamma \vdash E : M$, then $[\Gamma] \vdash [\overline{E}]: [M] / d^i(\Gamma) + \text{IsRec}(E)$.

**Proof** The proof is by structural induction on $E$, and case analysis on $E$.

**Function abstraction:** $E = \lambda x. C$ and $M = \tau_1 \rightarrow \tau_2$. By induction hypothesis, $[\Gamma] + \{ x : \tau_1 \} \vdash [C]: \tau_2 / d^i(\Gamma)[x \mapsto 0] + \text{IsRec}(C)$. Applying the degree weakening lemma if $\text{IsRec}(C)$ is not zero, we obtain $[\Gamma] + \{ x : \tau_1 \} \vdash [C]: \tau_2 / d^i(\Gamma)[x \mapsto 0]$. From this, the (abstr) typing rule shows that $[\Gamma] - [\lambda x. C] : \tau_1 \overset{\xi}{\rightarrow} \tau_2 / d^i(\Gamma) + 1$, which is the expected result since $\text{IsRec}(\lambda x. C) = 1$.

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

**Structure construction:** $E = \langle i; o \rangle$ and $M = \{ I; O; D \}$. By typing, we have $D = D\langle i; o \rangle, \vdash D, d\text{om}(o) = d\text{om}(O)$, and for all $X \in d\text{om}(o), \Gamma + I \circ o \vdash o(X) \vdash O(X)$.

Let $o = X_i \overset{i \xi}{\rightarrow} E_i, O = X_i \overset{i \xi}{\rightarrow} M_i, v_i = \text{IsRec}(E_i)$ and $\iota = y_j \overset{j \xi}{\rightarrow} Y_j$, with $I(Y_j) = M_j'$ for all $j$, with the $X_i$s and $Y_j$s ordered lexicographically, that is, if $i_1 < i_2$, then $X_{i_1} <_{\text{lex}} X_{i_2}$, and similarly for the $Y_j$s.

By induction hypothesis, for all $i$, we have $[\Gamma] + [I \circ o] \vdash [\overline{E_i}]: [M_i] / d^i(\Gamma + I \circ o) + v_i$.

But $\text{FV}([\overline{E}_i]) = \text{FV}(E_i)$ and $\text{FV}(E_i) \cap d\text{om}(i) = \iota \overset{i}{\rightarrow} (X_i)$, so we can apply lemma 35, and weakening lemmas 25 and 26 to eliminate variables of $d\text{om}(i)$ that are not free in $E_i$. Let $(Z_1, \ldots, Z_n) = D^{-1}(X_i)$ and for all $k \in \{ 1 \ldots n \}$, $M_k'' = I(Z_k)$. We obtain

\begin{align*}
\Gamma & \vdash \tilde{\lambda} \overset{i}{\rightarrow} (D^{-1}(X_i)), [\overline{E_i}]: [M_k''] \overset{\nu}{\rightarrow} \ldots [M_n''] \overset{\nu}{\rightarrow} [M_i] / d^i(\Gamma)
\end{align*}

But $\llbracket M_i \rrbracket_{X_i, D, \Gamma} = [M_i'] \overset{\nu}{\rightarrow} \ldots [M_n'] \overset{\nu}{\rightarrow} [M_i]$, because $D(Z_k, X_i) = \nu(\iota^{-1}(Z_k), E_i) = \text{IsRec}(E_i) = v_i$. The desired result follows.

**Closing**; $E = \text{close} E'$ and $M = \{ I; O; D \}$. We apply lemma 37 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 $D$, 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 $D$ would not be safe, and $M$ would be ill-formed). The expected result follows.

**Freezing**; $E = E_1 \times X$. The result follows from the induction hypothesis applied to $E_1$, and lemma 38 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 35 and 36 to handle the rearrangement of the parameters of the record components. \hfill \square

We define $\text{IsRec}(E)$ as $1$ if $E$ is an abstraction $\lambda x C$, and $0$ otherwise, and extend this definition to annotated expressions.
Theorem 4 (soundness of the translation) If $\Gamma \vdash E : M$, then $[\Gamma] \vdash [E] : [M] / d^e(\Gamma) + IsRec(E)$.

See appendix ?? for the full proof. Notice that this result holds for non-empty contexts $\Gamma$; 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.
Chapter 7

Compilation of let rec

7.1 Overview

The “in-place updating trick” The “in-place updating trick” outlined in [25] and refined in the OCaml compiler [55], implements let rec definitions that satisfy the following two conditions. Consider the mutually recursive definition \( x_1 = e_1 \ldots x_n = e_n \). First, the value of each definition should be represented at run-time by a heap allocated block of statically predictable size. Second, for each \( i \), the computation of \( e_i \) should not need the value of any of the definitions \( e_j \), but only their names \( x_j \). As an example of the second condition, a recursive definition like \( f = \lambda x. (\ldots f \ldots) \) is accepted, since no computation will try to use the value of \( f \). Contrarily, a recursive definition like \( f = (\lambda f. 0) \) is refused.

Evaluation of a let rec definition with in-place updating consists of three steps. First, for each definition, allocate an uninitialized block of the expected size, and bind it to the recursively-defined identifier. Those blocks are called dummy blocks. Second, compute the right-hand sides of the definitions. Recursively-defined identifiers thus refer to the corresponding dummy blocks. Owing to the second condition, no attempt is made to access the contents of the dummy blocks. This step leads, for each definition, to a block of the expected size. Third, the contents of the obtained blocks are copied to the dummy blocks, updating them in place.

For example, consider, in a given language \( L \), a mutually recursive definition \( x_1 = e_1, x_2 = e_2 \), where it is statically predictable that the values of the expressions \( e_1 \) and \( e_2 \) will be represented at runtime by heap allocated blocks of sizes \( n_1 \) and \( n_2 \), respectively. Here is what the compiled code does, as depicted in figure 7.1. First, it allocates two uninitialized heap blocks, at addresses \( l_1 \) and \( l_2 \), of sizes \( n_1 \) and \( n_2 \), respectively. This is called the pre-allocation step. As a second step, it computes \( e_1 \), where \( x_1 \) and \( x_2 \) are bound to \( l_1 \) and \( l_2 \), respectively. The result is a heap allocated block of size \( n_1 \), with possible references to the two uninitialized blocks. The same process is carried on for \( e_2 \), resulting in a heap allocated block of size \( n_2 \). The third and final step consists in copying the contents of the two obtained blocks to the two uninitialized blocks. The result is that the two initially dummy blocks now contain the proper cyclic data structure.

Simple generalization The scheme described above computes all definitions one after another, and only then updates the dummy blocks in place. From the example above, it seems quite clear that in-place updating for a definition could be done as soon as its value is available.

As long as mutual references do not really use the referenced values, as happens for recursive functions for instance, both schemes behave identically. Nevertheless, in the case where \( e_2 \) really uses the value \( n_1 \) computed for \( e_1 \), for example if \( e_2 = (x_1 \lambda) \), the original scheme can go wrong. Indeed, the dummy block pre-allocated for \( x_1 \) is still empty at the time where \( e_2 \) is computed. Instead,
with immediate in-place updating, the value \( v_1 \) is already available when computing \( e_2 \). This trivial modification to the scheme thus corresponds to increasing the expressive power of let rec. It allows definitions to really use previous definitions. Furthermore, it allows to transparently introduce definitions with unknown sizes in let rec, as shown by the following example.

An example of execution is presented in figure 7.2. The executed definition is \( x_1 = e_1, x_2 = e_2, x_3 = e_3 \), where \( e_1 \) and \( e_3 \) are expected to evaluate to blocks of sizes \( n_1 \) and \( n_3 \), respectively, but where the representation for the value of \( e_2 \) is not statically predictable. The pre-allocation step only allocates dummy blocks for \( x_1 \) and \( x_3 \). The value \( v_1 \) of \( e_1 \) is then computed. It can make references to \( x_1 \) and \( x_3 \), which correspond to pointers to the dummy blocks, but not to \( x_2 \), which would not make any sense here. This value is copied to the corresponding dummy block. Then, the value \( v_2 \) of \( e_2 \) is computed. It can refer to both dummy blocks, but it can also really use the value \( v_1 \). Finally, the value \( v_3 \) of \( e_3 \) is computed and copied to the corresponding dummy block.

This modified scheme implements more mutually recursive definitions than the initial one. The next section formalizes its semantics.

### 7.2 The source language \( \lambda_c \)

#### 7.2.1 Syntax

The syntax of \( \lambda_c \) is defined in figure 7.3. The meta-variables \( X \) and \( x \) range over names and variables, respectively. Variables are used as binders, as usual. Names are used for accessing record fields, as an external interface to other parts of the expression. Figure 7.4 recapitulates the meta-variables and notations we introduce in the remainder of this section. The syntax includes the \( \lambda \)-calculus constructs; variables \( x \), abstraction \( \lambda x. e \), and application \( e_1 e_2 \). The language also includes records \( \{X_1 = e_1 \ldots X_n = e_n\} \), record selection \( e.X \) and a let rec construct. A mutually
1. Pre-allocation:

2. Computing $e_1$:

3. Updating with $v_1$:

4. Computing $e_2$:

5. Computing $e_3$:

6. Updating with $v_3$:

Figure 7.2: The refined “in-place updating trick”
\[
\begin{align*}
  x & \in \text{Vars} & \text{Variable} \\
  X & \in \text{Names} & \text{Name} \\
  \text{Expression:} & \quad e \in \text{expr} ::= x | \lambda x. e | e_1 e_2 \\
  & \quad | \{ X_1 = e_1 \ldots X_n = e_n \} | e.X \\
  & \quad | \text{let rec } x_1 = e_1 \ldots x_n = e_n \text{ in } e
\end{align*}
\]

Figure 7.3: Syntax of \( \lambda_c \)

- More meta-variables:
  \[
  s ::= X_1 = e_1 \ldots X_n = e_n \quad \text{Record} \\
  b ::= x_1 = e_1 \ldots x_n = e_n \quad \text{Binding}
\]
- Notations:
  For a finite map \( f \), and a set of variables \( P \),
  \[
  \text{dom}(f) \text{ is its domain, } \quad \text{cod}(f) \text{ is its codomain} \\
  f_{|P} \text{ is its restriction to } P, \quad \text{and } f_{\setminus P} \text{ is its restriction to } \text{Vars}\setminus P.
\]
- Expressions of predictable shape:
  \[
  e_4 \in \text{Predictable ::= } \{@\} | (@;@) | \text{let rec } b \text{ in } e_4
\]

Figure 7.4: Meta-variables and notations

A recursive definition has the shape let rec \( x_1 = e_1 \ldots x_n = e_n \) in \( e \), where arbitrary expressions are syntactically allowed as the right-hand side of a definition.

**Syntactic correctness** Records \( s = (X_1 = e_1 \ldots X_n = e_n) \) and bindings \( b = (x_1 = e_1 \ldots x_n = e_n) \) are required to be finite maps: a record is a finite map from names to expressions, and a binding is a finite map from variables to expressions. Requiring them to be finite maps means that they should not bind the same variable or name twice.

Consider the let rec binding \( b = (x_1 = e_1 \ldots x_n = e_n) \). We say that there is a forward reference from \( x_i \) to \( x_j \) if \( i \leq j \), and \( x_j \) occurs free in \( e_i \).

Forward references in bindings are allowed only when they point to a certain class of expressions, the expressions of predictable shape. As a first approximation, we say that the shape of an expression is predictable if it is a structure, a record, or a binding followed by an expression of predictable shape. Formally \( e_4 \in \text{Predictable ::= } \{\cdot\} | (@;\cdot) | \text{let rec } b \text{ in } e_4 \).

**Sequences** Records and bindings are often considered as finite maps in the sequel. We refer to them collectively as sequences, and use the usual notions on finite maps, such as the domain \( \text{dom} \), the codomain \( \text{cod} \), the restriction \( \cdot_{|P} \) to a set \( P \), or the co-restriction \( \cdot_{\setminus P} \) outside of a set \( P \).

### 7.2.2 Structural equivalence

We consider the expressions equivalent up to alpha-conversion of binding variables in structures and let rec expressions. For this, we define the structural contraction relation, in figure 7.8, relying on notions defined just below.

A binder \( x \), in a let rec or in a function, may be renamed into a new variable \( y \), provided \( y \) meets some freshness conditions. Variable renaming is formally defined in figure 7.7, using notions defined in figures 7.5 and 7.6. Variable renaming is a total function, from pairs of an expression and a
\[
\text{UnsafeNewNames}(x, \lambda x.e) = \text{Cpt}_x(e) \cup \text{FV}(e)
\]
\[
\text{UnsafeNewNames}(x, \text{let rec } b \text{ in } e) = \left( \text{FV}(\text{let rec } b \text{ in } e) \cup \text{Cpt}_x(e) \right) \bigcup \left( \bigcup_{y \in \text{dom}(b)} \left( \{y\} \cup \text{Cpt}_x(b(y)) \right) \right) \cup \{x\}
\]

Figure 7.5: Unsafe new names in \( \lambda_c \)

\[
\text{Cpt}_x(\text{let rec } b \text{ in } e) = \begin{cases} 
\bigcup_{y \in \text{dom}(b)} \left( \{y\} \cup \text{Cpt}_x(b(y)) \right) & \text{if } x \in \text{FV}(\text{let rec } b \text{ in } e) \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
\text{Cpt}_x(x) = \text{Cpt}_x(e) = \emptyset
\]

\[
\text{Cpt}_x(\lambda y.e) = \begin{cases} 
\{y\} \cup \text{Cpt}_x(e) & \text{if } x \in \text{FV}(\lambda y.e) \\
\emptyset & \text{otherwise}
\end{cases}
\]

Figure 7.6: Capture in \( \lambda_c \)

variable renaming \( x \mapsto y \) (\( x \) is replaced with \( y \)), to expressions. In case renaming crosses a node binding one of the two variables \( x \) and \( y \), it stops. Otherwise, it is propagated as usual. Therefore, variable renaming sometimes does not preserve meaning. For instance, renaming \( x \) with \( y \) in \( \lambda y. x \) yields the same expression, since renaming does not cross the node binding \( y \). This is why we introduce the notion of unsafe new names. It is defined in figure 7.5. A new name can be unsafe for a binder if it is captured by binders inside the sub-expression, as \( y \) is in the above example. The notion of capture is formalized by the \( \text{Cpt} \) function in figure 7.6. Basically, \( \text{Cpt}_x(e) \) denotes the set of binding variables located above occurrences of \( x \) in \( e \). For instance \( \text{Cpt}_x(\lambda y.x) \) is the set \( \{y\} \). A new name can also be unsafe for a binder when it is free in the considered sub-expression. For example, renaming \( x \) to \( y \) in \( \lambda x. (xy) \) does not preserve meaning. The structural contraction relation, \( == \), defined in figure 7.8, allows to rename a binder, provided the corresponding variable renaming is correct on the considered expression. The structural reduction relation \( \longrightarrow^*_s \) is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure \( \longrightarrow^*_s \) of \( \longrightarrow^*_s \) is a congruence, called the structural equivalence relation, and also written \( =_s \).

In the following, all expressions are considered up to structural equivalence \( =_s \).

Let \( \sigma = \{x \mapsto y\} \).

\[
\begin{align*}
x(\sigma) &= y \\
z(\sigma) &= z \text{ if } z \neq x \\
\{X_1 = e_1 \ldots X_n = e_n\}(\sigma) &= \{X_1 = e_1\{\sigma\} \ldots X_n = e_n\{\sigma\}\} \\
(\lambda z.e)(\sigma) &= \begin{cases} 
\lambda z.(e(\sigma)) & \text{if } z \notin \{x,y\} \\
\lambda z.e & \text{otherwise}
\end{cases} \\
(\text{let rec } b \text{ in } e)(\sigma) &= \begin{cases} 
\text{let rec } b(\sigma) \text{ in } e(\sigma) & \text{if } \{x,y\} \cap \text{dom}(b) = \emptyset \\
\text{let rec } b \text{ in } e & \text{otherwise}
\end{cases} \\
(x_1 = e_1 \ldots x_n = e_n)(\sigma) &= (x_1 = e_1(\sigma) \ldots x_n = e_n(\sigma))
\end{align*}
\]

Other cases easy.

Figure 7.7: Variable renaming in \( \lambda_c \)
$$y \notin \text{UnsafeNewNames}(x, \text{let rec } b_1, x = e, b_2 \text{ in } f) \quad \sigma = x \mapsto y$$
$$\text{let rec } b_1, x = e, b_2 \text{ in } f \rightsquigarrow_s \text{let rec } b_1 (\sigma), y = e (\sigma), b_2 (\sigma) \text{ in } f (\sigma)$$

$$y \notin \text{UnsafeNewNames}(x, \lambda x.e)$$
$$\lambda x.e \rightsquigarrow_s \lambda y. (e (x \mapsto y))$$

Figure 7.8: Structural contraction relation of $\lambda_c$

Configuration: $e ::= b \vdash e$
Value: $v \in \text{values} ::= x \mid \lambda x.e \mid \{s_e\}$
Answer: $a \in \text{answers} ::= b_e \vdash v$

More meta-variables:
$s_v ::= X_1 = v_1 \ldots X_n = v_n$ Value record
$b_v ::= x_1 = v_1 \ldots x_n = v_n$ Value binding

Figure 7.9: Configurations and results in $\lambda_c$

7.2.3 Semantics

The semantics of $\lambda_c$ is quite standard, except for what concerns let rec bindings.

As shown in figure 7.9, values include functions $\lambda x.e$ and records of values $\{s_e\}$, where $s_e$ denotes an evaluated record $X_1 = v_1 \ldots X_n = v_n$.

The semantics of record selection and of function application are defined in figure 7.10, by computational contraction rules, defining the local computational contraction relation $\rightsquigarrow_s$. Record projection selects the appropriate field in the record; and the application of a function $\lambda x.e$ to a value $v$ reduces to the body of the function, where the argument has been bound to $x$.

Five operations are necessary for handling bindings properly, all defined Ariola et al. [7].

1. A first operation is let rec lifting. It consists in lifting a let rec node up one level in an expression. For example, an expression of the shape $e_1 + (\text{let rec } b \text{ in } e_2)$ becomes let rec $b \text{ in } e_1 + e_2$.

2. A second operation is internal merging. During the evaluation of a binding, a definition may return a let rec as an answer, where a value is expected. Internal merging merges this binding into the current one. An expression of the shape let rec $b_1, x = (\text{let rec } b_2 \text{ in } e), b_3 \text{ in } f$ becomes let rec $b_1, b_2, x = e, b_3 \text{ in } f$, provided no variable capture occurs.

3. A third operation is external merging. The shape of results in $\lambda_c$ allows only one binding to wrap values. Therefore, if evaluation results in two nested bindings, they must be merged into a single one. An expression of the shape let rec $b_1 \text{ in } \text{let rec } b_2 \text{ in } e$ becomes let rec $b_1, b_2 \text{ in } e$, provided no variable capture occurs.

4. A fourth operation, external substitution, allows to access bound variables when defined by a surrounding binding. An expression of the shape let rec $b \text{ in } C [x]$ becomes let rec $b \text{ in } C [x]$, if $x = e$ appears in $b$ and $x$ is not captured by $C$, and no variable capture occurs.

5. A last operation, internal substitution, allows to access identifiers bound earlier in the same binding. (Assuming left-to-right evaluation, “earlier” means “to the left of”.) An expression of the shape let rec $b_1, y = C [x], b_2 \text{ in } e$ becomes let rec $b_1, y = C [f], b_2 \text{ in } e$ if $x$ is defined as $f$ in $b_1$, and not captured by $C$, and no variable capture occurs.
### Contraction rules

\[\{X_1 = v_1 \ldots X_n = v_n\}.X_i \sim_e v_i \quad \text{(PROJECT)} \quad x \notin FV(v)\]

\[\frac{}{(\lambda x.e)v \sim_e \text{let rec } x = v \text{ in } e} \quad \text{(BETA)}\]

\[\text{dom}(b) \perp FV(L)\]

\[\frac{\text{L} \downarrow \text{let rec } b \in e \sim_e \text{let rec } b \in \text{L}[e]}{\text{L}[e]} \quad \text{(LIFT)}\]

### Computational reduction rules

\[
\frac{e \sim_e e'}{E[e] \rightarrow^e E[e']} \quad \text{(CONTEXT)} \quad \frac{E[N](x) = v}{E[N[x]] \rightarrow^e E[N[v]]} \quad \text{(SUBST)}
\]

\[\frac{\text{dom}(b) \perp (\text{dom}(b_v) \cup FV(b_v))}{(b_v \vdash \text{let rec } b \in e) \sim_e (b_v, b \vdash e)} \quad \text{(EM)}\]

\[\frac{\text{dom}(b_1) \perp \{x\} \cup \text{dom}(b_1, b_2) \cup FV(b_1, b_2) \cup FV(f)}{(b_1, x = (\text{let rec } b_1 \in e), b_2 \vdash f) \sim_e (b_1, b_1, x = e, b_2 \vdash f)} \quad \text{(IM)}\]

### Evaluation contexts

Lift context:
\[
\begin{align*}
L &:= \Box \circ | v \circ | \Box.X | \{S\} \\
S &:= s_e.X = \Box, s
\end{align*}
\]

Multiple lift context:
\[
F := \Box | L[F] \quad B := b_v, x = \Box, b
\]

Evaluation context:
\[
E := (b_v \vdash F) | (B[F] \vdash e) \quad N := \Box v | \Box.X
\]

### Access in evaluation contexts

\[
(b_v \vdash F)(x) = b_v(x) \quad \text{(EA)} \quad (b_v, y = F, b \vdash e)(x) = b_v(x) \quad \text{(IA)}
\]

**Figure 7.10: Reduction semantics for \(\lambda_e\)**

The question is how to arrange these operations to make the evaluation deterministic and to ensure that it reaches the result when it exists. Our choice can be summed up as follows. There is a topmost binding. When this topmost binding is already evaluated, evaluation can proceed under this binding. Otherwise, evaluation is allowed inside this binding. If evaluation meets another binding inside the expression, this binding is lifted to be immediately under the topmost binding. Then, it is merged with the latter, internally or externally according to the context. External and internal substitutions are allowed only from the evaluated part of the topmost binding. In order to simplify the presentation of the translation and the correctness proof, we distinguish this topmost binding syntactically: the global computational reduction relation \(\rightarrow^e\), is a binary relation on configurations \(c\), which are pairs of a binding, the topmost binding, and an expression, written \(b \vdash e\) (see figure 7.9). Here, the topmost binding is close to the usual notion of runtime environment, with the additional feature that bound values can be mutually recursive.

More formally, let rec handling is done through one additional computational contraction rule \(\text{LIFT}\) performing the lifting operation, and a computational reduction relation, defined in figure 7.10.

The contraction rule \(\text{LIFT}\) lifts a let rec binding up a lift context. As defined in figure 7.10, a lift context is any non-let rec expression, where the special context hole variable \(\Box\) appears immediately under the first node, in position of the next sub-expression evaluated.

The second contraction rule \(\text{IM}\) corresponds to internal merging. If, during the evaluation of the topmost binding, one definition evaluates to a binding, then this binding is merged with the
\[ x \in \text{Vars} \]
\[ X \in \text{Names} \]

Expression:
\[ E :\text{Expr} ::= x \mid \lambda x. E \mid EE \]
\[ \text{\text{\text{\text{\text{\text{\text{\text{$\lambda$-calculus}}}}}}}} \]
\[ \text{\text{\text{\text{\text{\text{\text{\text{let}}}}}}}}\ x_1 = E_1 \ldots x_n = E_n\ \text{in}\ E \]
\[ \text{\text{\text{\text{\text{\text{\text{\text{Non-recursive let binding}}}}}}}} \]
\[ \{ X_1 = E_1 \ldots X_n = E_n \} \mid E. X \]
\[ \text{\text{\text{\text{\text{\text{\text{\text{Records}}}}}}}} \]
\[ I \mid \text{alloc update} \]
\[ \text{\text{\text{\text{\text{\text{\text{\text{Locations, allocation, mutation}}}}}}}} \]

Figure 7.11: Syntax of $\lambda_{alloc}$

topmost one. The evaluation can then continue.

The computational reduction relation extends the computational contraction relation to any evaluation context, as defined in figure 7.10. We call a multiple lift context a series of nested lift contexts, and an evaluation context is a multiple lift context, possibly inside a partially evaluated binding, or under a fully evaluated binding.

The EM reduction rule corresponds to external merging. It is only possible at toplevel, provided no variable capture occurs.

Finally, the external and internal substitution operations are modeled within a single reduction rule $\text{SUBST}$. This rule transforms an expression of the shape $E[N[x]]$ into $E[N[v]]$, provided the context $E[N]$ defines $x$ as $v$ and no variable capture occurs. The meta-variable $N$ ranges over strict contexts. A strict context is a context that requires a non-variable node to evaluate. An example of a strict context is $\square v$, that is, the function part of a function application. An example of a non-strict context is $(\lambda x.e)[\square]$, that is, the argument part of a function application, where a variable would allow the evaluation to continue. Strict contexts are formally defined in figure 7.10. The $\text{SUBST}$ rule replaces a variable in a strict context with its value, according to the context. As indicated in figure 7.10, evaluation contexts define the variable they bind, in two possible ways. First, a topmost, semantically correct, fully evaluated let rec binding defines the variables it binds for the nodes under it. Second, if $(b_v, x \in F, b)$ is the topmost, partially evaluated binding, then $b_v$ defines the variables it binds, inside $F$, and later inside $b$. The two rules defining access in evaluation contexts in figure 7.10 show how these definitions may be used. The two different ways of access correspond to the external and internal substitution operations, respectively.

The computational reduction relation on expressions is compatible with structural equivalence $\equiv_s$. Hence we can define computational reduction over equivalence classes of expressions, obtaining the reduction relation $\rightarrow_s$.

### 7.3 The target language $\lambda_{alloc}$

The syntax of the target language $\lambda_{alloc}$ is presented in figure 7.11. It distinguishes variables $x$ from names $X$. It includes the constructs of the $\lambda$-calculus (function abstraction and application) and a non-recursive let binding. Additionally, there are constructs for record operations (construction and selection), and constructs for modeling the heap: an allocation operator $\text{alloc}$, an update operator $\text{update}$, and locations $l$.

The semantics of $\lambda_{alloc}$ is defined as a structural reduction relation on configurations. As defined in figure 7.12, a configuration is a pair of a heap and an expression. A heap is a finite map from locations $l$ to evaluated heap blocks. An evaluated heap block $H_v \in \text{HeapValues}$ is either a function $\lambda x. E$, or an evaluated record $\{ S_v \}$ (where $S_v ::= X_1 = V_1 \ldots X_n = V_n$), or an application of the shape $\text{alloc} n$, for $n \in \mathbb{N}$. Such applications model dummy heap blocks, containing unspecified data. A well-formed configuration is such that all the locations mentioned are bound in its heap.
Configuration:

\[

c \ ::= \Theta \vdash E \\
\Theta \in \text{Heaps} = \text{Vars} \xrightarrow{\text{Fin}} \text{HeapValues}
\]

Answer:

\[
A \in \text{Answers} ::= \Theta \vdash V \\
V \in \text{Values} ::= x \mid l
\]

More meta-variables:

\[
H_v \in \text{HeapValues} ::= \lambda x.E \mid \text{alloc } n \mid \{S_v\}
\]

\[
S_v ::= X_1 = V_1 \ldots X_n = V_n
\]

\[
B ::= x_1 = E_1 \ldots x_n = E_n
\]

Figure 7.12: Configurations and results in $\lambda_{alloc}$

Evaluated heap blocks are not values. Only variables and locations are values. In this calculus, function abstractions are not values, since their evaluation allocates the function in the heap, and returns its location: the result of the evaluation of $\lambda x.E$ is a configuration $\Theta \vdash l$, where the location $l$ is bound to $\lambda x.[c]$ in the heap $\Theta$.

The related operators in the language are alloc, which creates a new empty block of size given by its argument, and update, which copies its second argument in place of its first one, provided they have the same size. For this, we assume given a function $\text{Size}$ from $\lambda_{alloc}$ heap value blocks to $\mathbb{N}$.

**Notation** We write $\Theta(l \mapsto H_v)$ for the map equal to $\Theta$ anywhere but on $l$ where it returns $H_v$.

We write $\Theta_1 + \Theta_2$ for the union of two heaps $\Theta_1$ and $\Theta_2$ whose domains are disjoint. In particular, when the heap $\Theta$ is undefined on $l$, we write $\Theta + \{l \mapsto H_v\}$ to denote the union of $\Theta$ and $\{l \mapsto H_v\}$.

### 7.3.1 Structural equivalence

In $\lambda_{alloc}$, a notion of structural equivalence identifies expressions modulo variable and location renaming. Locations are bound only by heaps, at toplevel in configurations. We consider configurations equal modulo renaming of bound locations. This relation is easy to define since the location renaming never cross any location binder, so we do not formalize it here. However, we have to define the structural equivalence modulo variable renaming. A binder $x$, in a let or in a function, may be renamed into a new variable $y$, provided $y$ meets some freshness conditions. Structural equivalence is formally defined in figure 7.13.

**Substitutions** First, variable renaming is defined. It is a total function, from pairs of an expression and a variable renaming $x \mapsto y$ ($x$ is replaced with $y$), to expressions. Nevertheless, we will see that the computational reduction relation uses a more complex notion of substitution than just variable renaming: it must also replace variables with locations in some cases. Therefore, substitutions are elements of $\text{Subst} = \text{Vars} \xrightarrow{\text{Fin}} \text{Values}$. We interpret them as total functions from variables to values, extending them with the identity function on variables, outside of their syntactic domain. The domain $\text{dom}(\sigma)$ of a substitution $\sigma$ is the set of variables $x$ such that $\sigma(x) \neq x$.

We sometimes consider substitutions as sets, taking the union of two of them when it makes sense, and sometimes we compose them, in the reverse notation, since they come from the right. The composition of $\sigma_1$ and $\sigma_2$ is defined by $x\{\sigma_1 \circ \sigma_2\} = x \{\sigma_1\} \{\sigma_2\}$: it acts as $\sigma_1$, then $\sigma_2$. Moreover, we call variable renamings, or simply renamings, the injective substitutions whose codomains contain
**Substitutions** Let $\sigma \in \text{Subst} = \text{Vars} \xrightarrow{\text{Fin}} \text{Values}$.

\[
\begin{align*}
z\{\sigma\} & = \sigma(z) \\
(\lambda x. E)\{\sigma\} & = \lambda x. (E[\sigma \setminus \{x\}]) \\
(\text{let } B \text{ in } F)\{\sigma\} & = \text{let } B[\sigma] \text{ in } F[\sigma \setminus \text{dom}(B)] \\
(x = E, B)\{\sigma\} & = (x = E[\sigma], B[\sigma \setminus \{x\}]) \\
\Theta \vdash E\{\sigma\} & = \sigma \circ \Theta \vdash E\{\sigma\}
\end{align*}
\]

Other cases easy.

**Capture**

\[
\begin{align*}
\text{Capt}_x(\text{let } y = E, B \text{ in } F) & = \text{Capt}_x(\lambda y. \text{let } B \text{ in } F) \cup \text{Capt}_x(E) \\
\text{Capt}_x(x) & = \emptyset \\
\text{Capt}_x(\lambda y. E) & = \begin{cases} 
\{y\} \cup \text{Capt}_x(E) & \text{if } x \in \text{FV}(\lambda y. E) \\
\emptyset & \text{otherwise}
\end{cases}
\end{align*}
\]

Other cases easy.

**Unsafe new names**

\[
\begin{align*}
\text{UnsafeNewNames}(x, \lambda x. E) & = \text{Capt}_x(E) \cup \text{FV}(E) \\
\text{UnsafeNewNames}(x, \text{let } x = E, B \text{ in } F) & = \text{UnsafeNewNames}(x, \lambda x. (\text{let } B \text{ in } F))
\end{align*}
\]

**Structural reduction**

\[
\begin{align*}
y \notin \text{UnsafeNewNames}(x, \lambda x. E) \\
\lambda x. E \leadsto_s \lambda y. (E\{x \mapsto y\})
\end{align*}
\]

\[
\begin{align*}
y \notin \text{UnsafeNewNames}(x, \text{let } x = E, B \text{ in } F) \\
\text{let } x = E, B \text{ in } F \leadsto_s \text{let } y = E, (B\{x \mapsto y\}) \text{ in } F\{x \mapsto y\}
\end{align*}
\]

Figure 7.13: Structural equivalence in $\lambda_{alloc}$
\[ \Theta(l) = \lambda x. E \]
\[ \Theta \vdash lV \leadsto_e \Theta \vdash E[x \mapsto V] \quad \text{(BETA)} \]
\[ l \not\in \text{dom}(\Theta) \]
\[ \Theta \vdash H_e \leadsto_e \Theta + [l \mapsto H_e] \vdash t \quad \text{(ALLOCATE)} \]
\[ \Theta(l) = \{S_e\} \]
\[ \Theta \vdash l.X \leadsto_e \Theta + S_e(X) \quad \text{(PROJECT)} \]
\[ \text{Size}(\Theta(l_1)) = \text{Size}(\Theta(l_2)) \]
\[ \Theta \vdash \text{update} \ l_1 \ l_2 \leadsto_e \Theta (l_1 \mapsto \Theta(l_2)) \vdash \{ \} \quad \text{(UPDATE)} \]
\[ \text{dom}(B) \perp \Lambda \]
\[ \Theta \vdash \Lambda[\text{let } B \text{ in } E] \leadsto_e \Theta \vdash \text{let } B \text{ in } \Lambda[E] \quad \text{(LIFT)} \]

Figure 7.14: Computational contraction rules for \(\lambda_{\text{alloc}}\)

only variables, and we denote them by \(\zeta\). Symmetrically, we call variable allocations the injective substitutions mapping variables to locations, and denote them by \(\eta\).

We extend substitutions to \(\lambda_{\text{alloc}}\) expressions and configurations, as described in figure 7.13 (where we take the usual notation for substitution \(E[\sigma]\), meaning \(\sigma(E)\)). In case the substitution crosses a binder \(x\), then it forgets any information about \(x\). Thus, under this binder the substitution becomes \(\sigma \setminus \{x\} \cup \sigma^{-1}(\{x\})\). Otherwise, it is propagated as usual. Therefore, substitution sometimes does not preserve meaning. For instance, renaming \(x\) with \(y\) in \(\lambda y.x\) yields the same expression, since substitution does not cross the node binding \(y\).

**Structural equivalence** This is why we introduce the notion of unsafe new names. It is defined in figure ???. A new name can be unsafe for a binder if it is captured by binders inside the sub-expression, as \(y\) in the above example. The notion of capture is formalized by the \(\text{Capt}\) function in figure 7.13. Basically, \(\text{Capt}_x(e)\) denotes the set of binding variables located above occurrences of \(x\) in \(e\). For instance \(\text{Capt}_x(\lambda y.x)\) is the set \(\{y\}\). A new name can also be unsafe for a binder when it is free in the considered sub-expression. As an example, renaming \(x\) to \(y\) in \(\lambda x.(xy)\) does not preserve meaning.

The structural contraction relation, \(\leadsto_s\), defined in figure 7.13, allows to rename a binder, provided the corresponding variable renaming is correct on the considered expression. The structural reduction relation \(\rightarrow_s\) is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure \(\rightarrow_s^*\) of \(\rightarrow_s\) is a congruence, called the structural equivalence relation, and also written \(=_s\).

### 7.3.2 Semantics

The semantics of \(\lambda_{\text{alloc}}\), like the one of \(\lambda_e\), is given in terms of a computational contraction relation that handles rules for the basic constructors and a computational reduction relation that handles global rules. As in \(\lambda_e\), evaluation results are values surrounded by a heap binding:

\[ A \in \text{Answers} ::= \Theta \vdash V. \]

**Computational contraction relation** The computational contraction relation is defined by the rules in figure 7.14, using the notion of lift contexts in figure 7.15.

The BETA rule is a bit unusual, in that it applies a heap allocated function to an argument \(V\). The function must be a heap binding \(l \mapsto \lambda x. E\), and the result is \(E[x \mapsto V]\).
Lift context:
\[ \Lambda ::= \Box E | V \Box | \Box X | \{ \Sigma \} \mid \text{let } x = \Box, B \text{ in } e \]

Record context:
\[ \Sigma ::= S_e, X = \Box, S \]

Multiple lift context:
\[ \Phi ::= \Box | \Lambda[\Phi] \]

---

**Figure 7.15:** Evaluation contexts of λ_{alloc}

\[
\frac{\Theta \vdash E \rightsquigarrow_{e} \Theta' \vdash E'}{\Theta \vdash \Phi[E] \rightsquigarrow_{e} \Theta' \vdash \Phi[E']} \quad \text{(CONTEXT)}
\]

\[
\Theta \vdash \text{let } x = V, B \text{ in } E \rightsquigarrow_{e} \Theta \vdash (\text{let } B \text{ in } E) \{x \mapsto V\} \quad \text{(LET)}
\]

\[
\frac{\Theta \vdash \text{let } e \text{ in } E \rightsquigarrow_{e} \Theta \vdash E}{l \notin \text{FV}(\Theta[\{l\}]) \cup \text{dom}(\Theta[\{l\}]) \cup \text{FV}(E)} \quad \text{(GC)}
\]

\[
\Theta \vdash \text{let } B_1 \text{ in } \text{let } B_2 \text{ in } E \rightsquigarrow_{e} \Theta \vdash \text{let } B_1, B_2 \text{ in } E \quad \text{(EM)}
\]

---

**Figure 7.16:** Computational reduction in λ_{alloc}

The **PROJECT** rule works similarly: it projects a name \( X \) out of a heap allocated record \( l \mapsto \{S_e\} \), where \( S_e \) is a finite set of evaluated record field definitions of the shape \( X_1 = V_1 \ldots X_n = V_n \). The result is \( S_e(X) \) (i.e. \( V_i \) is \( X_i \)).

The **ALLOCATE** rule is one of the key points of λ_{alloc}. It states that a value block \( H_e \) evaluates into a fresh heap location containing \( H_e \), and a pointer to it: \( \Theta + \{l \mapsto H_e\} \vdash l \) (\( l \) fresh). If \( H_e \) is a dummy block alloc\( \n \), the result is a dummy block on the heap.

The **UPDATE** rule copies the contents of a heap block on to another one. If the locations \( l_1 \) and \( l_2 \) are respectively bound to blocks \( H_{e1} \) and \( H_{e2} \) in the heap \( \Theta \), then \( \Theta \vdash \text{update} l_1 l_2 \) will evaluate to \( \Theta[l_1 \mapsto H_{e2}] \vdash \{l_1 \mapsto \} \).

Finally, as in \( \lambda_c \), the evaluation of bindings is confined to the toplevel of terms, whence the **LIFT** rule, which lifts a binding outside of a lift context. In \( \lambda_{alloc} \), lift contexts are of the shape

\[ \Lambda ::= \Box E | V \Box | \Box X | \{ \Sigma \} \mid \text{let } x = \Box, B \text{ in } e, \]

where \( \Sigma \) ranges over record contexts, of the shape \( \Sigma ::= S_e, X = \Box, S \).

**Computational reduction relation** The computational reduction relation is defined in figure 7.16.

The **CONTEXT** rule shifts the contraction relation to a multiple lift context. Lift contexts have been defined in the last paragraph, and multiple lift contexts are simply series of nested lift contexts.

The **LET** rule describes the toplevel evaluation of bindings. Once the first definition is evaluated, the binding variable is replaced with the obtained value in the rest of the expression. Eventually, when the binding is empty, it can be removed with rule **EMPTYLET**.

By rule **GC**, when a heap binding is not used by any other binding than itself, and not used either by the expression, it may be removed.

Finally, the **EM** rule states that it is equivalent to evaluate two bindings in succession, or to evaluate their union.

166
7.3.3 The $\lambda_{alloc}$ calculus and its confluence

The set of terms of the $\lambda_{alloc}$ calculus is the set of equivalence classes for $=_{s}$. The computational reduction relation on expressions is compatible with $=_{s}$, so we may extend it to terms, to obtain the reduction relation $\rightarrow$.

**Definition 19** The $\lambda_{alloc}$ calculus is the set of terms, equipped with the relation $\rightarrow$.

Unlike in $\lambda_{c}$, the reduction of $\lambda_{alloc}$ is not deterministic because of rules GC and EM. Rule GC can apply at any time, and rule EM gives a choice between two outcomes when two successive bindings are encountered. It is therefore important to make sure that $\lambda_{alloc}$ is confluent. Let $\xrightarrow{\text{C}}$ be the relation defined by the rules CONTEXT, LET, and EMPTYLET. It is syntax directed, and therefore deterministic.

We first prove the following proposition, which is also described by the following diagram, where the plain arrows are universally quantified, and the dotted ones are existentially quantified.

```
\[
\begin{array}{ccc}
C & \xrightarrow{c} & C_1 \\
\downarrow & & \downarrow \text{EM} \\
C_1 & & C_2 \\
\end{array}
\]
```

**Proposition 9** For all configurations $C$, $C_1$, and $C_2$ such that $C \xrightarrow{c} C_1$ and $C \xrightarrow{\text{EM}} C_2$, there exists a configuration $C'$ such that $C_1 \xrightarrow{\text{EM}} C'$ and $C_2 \xrightarrow{c} C'$.

**Proof**

If $C \xrightarrow{\text{EMPTYLET}} C_1$, the two obtained configurations are identical. If $C \xrightarrow{c} C_1$ by rule LET, then the two reductions simply commute. If $C \xrightarrow{\text{CONTEXT}} C_1$, then we have to examine the underlying contraction step $C \rightarrow_{c} C_1$. In all cases but one, the two reduction steps simply commute. The only problematic case is when the applied rule is LIFT. We have $C = \Theta \vdash \Phi \{E\}$, with $E = \Lambda[\text{let } B \text{ in } E_1]$, and $C_1 = \Theta \vdash \Phi[\text{let } B \text{ in } \Lambda[E_1]]$.

- If $\Lambda ::= \square F \mid \square \mid \{\Sigma\} \mid \square X$, as rule EM applies on $C$, we must have $\Phi$ of the shape $\lambda x = \Phi_1, B_1$ in let $B_2$ in $F'$. Therefore

  \[
  C_1 = \Theta \vdash \text{let } x = \Phi_1, B_1 \text{ in } \Lambda[E_1], B_1 \text{ in } B_2 \text{ in } F',
  \]

  and

  \[
  C_2 = \Theta \vdash \text{let } x = \Phi_1[\Lambda[\text{let } B \text{ in } E_1]], B_1, B_2 \text{ in } F'.
  \]

Let

\[
C' = \Theta \vdash \text{let } x = \Phi_1[\text{let } B \text{ in } \Lambda[E_1]], B_1, B_2 \text{ in } F'.
\]

We obtain easily that $C_1$ and $C_2$ both reduce to $C'$, in one step of $\xrightarrow{\text{EM}}$ and $\xrightarrow{\text{LIFT}}$, respectively, which is as expected.
• If $\Lambda = \text{let } x = \Box, B_1 \text{ in } F$, then $\Phi$ might still be of the shape $\text{let } x = \Phi_1, B'_1 \text{ in } \text{let } B_2 \text{ in } F''$, in which case the previous reasoning applies. If it is not of this shape, then the let binding contained in $\Lambda$ is part of the EM redex, so $\Phi = \Box$, and $F$ is of the shape $\text{let } B_2 \text{ in } F''$. So, we have a diagram of the shape:

$$\Theta \vdash \text{let } x = (\text{let } B \text{ in } E_1), B_1 \text{ \textbf{in} } F''$$

$\Theta \vdash \text{let } B_1, B_2 \text{ \textbf{in} } F''$

$\Theta \vdash \text{let } B \text{ \textbf{in} } E_1, B_1 \text{ \textbf{in} } F''$  \hspace{1cm} $\Theta \vdash \text{let } B \text{ \textbf{in} } E_1, B_1 \text{ \textbf{in} } F''$

- $\Theta \vdash \text{let } x = E_1, B_1 \text{ \textbf{in} } F''$

- $\Theta \vdash \text{let } B, x = E_1, B_1 \text{ \textbf{in} } F''$

$\Theta \vdash \text{let } B, x = E_1, B_1, B_2 \text{ \textbf{in} } F''$

This result extends by a simple induction to the following corollary, pictorially described by the following diagram.

![Diagram](image)

**Corollary 7** For all configurations $C$, $C_1$, and $C_2$ such that $C \xrightarrow{C} C_1$ and $C \xrightarrow{\text{EM}*} C_2$, there exists a configuration $C'$ such that $C_1 \xrightarrow{\text{EM}*} C'$ and $C_2 \xrightarrow{C, \text{EM}*} C'$.

Then, the relation $\xrightarrow{\text{EM}}$ is defined as $\xrightarrow{C}$, extended with rule EM. Formally, $\xrightarrow{\text{EM}} = \xrightarrow{C} \cup \xrightarrow{\text{EM}}$.

Thanks to the previous corollary, we prove that the $\xrightarrow{\text{EM}}$ relation is confluent. This is done by considering the relation $\xrightarrow{\text{EM} \text{ EM}*}$, which is strongly confluent. In other terms for any two reduction steps $C \xrightarrow{\text{EM} \text{ EM}*} C_1$ and $C \xrightarrow{\text{EM} \text{ EM}*} C_2$, there exist a configuration $C'$ and two reduction steps...
\[ C_1 \xrightarrow{\text{CEM EM}^*} C' \quad \text{and} \quad C_2 \xrightarrow{\text{CEM EM}^*} C'. \] A pictorial view of this is given by the following diagram:

![Diagram](image)

**Proposition 10** The relation \( \xrightarrow{\text{CEM EM}^*} \) is strongly confluent.

**Proof** To prove this last statement, we proceed by case on the CEM rules applied, from \( C \), to reach \( C_1 \) and \( C_2 \), respectively. If the two rules are EM, then as this relation is deterministic, we conclude easily, and similarly if the two reductions are \( \xrightarrow{c} \) steps. The only relevant case is when one reduction is a \( \xrightarrow{c} \) step, say \( C \xrightarrow{c} C_1 \), and the other is in \( \xrightarrow{\text{EM}} \).

In this case, we have \( C \xrightarrow{c} C'_1 \xrightarrow{\text{EM}^*} C_1 \). By the previous corollary, we obtain a \( C'_2 \) such that \( C_2 \xrightarrow{\text{CEM EM}^*} C'_2 \). Then, by confluence of the deterministic relation \( \xrightarrow{\text{EM}} \), we obtain \( C' \) such that \( C'_1 \xrightarrow{\text{EM}^*} C' \) and \( C'_2 \xrightarrow{\text{EM}^*} C' \). This configuration is also such that \( C_1 \) and \( C_2 \) reduce to it by relation \( \xrightarrow{\text{CEM EM}^*} \), in at most one step.

This is depicted by the following diagram.
Corollary 8 (Confluence of $\lambda_{alloc}$) The $\lambda_{alloc}$ calculus is confluent.

7.4 Translation

7.4.1 Generalized contexts in $\lambda_{alloc}$

The purpose of this paper is to prove that $\lambda_c$ can be faithfully translated into $\lambda_{alloc}$. A desired property for this translation, in order to make the proof of correctness easier, is that a result is translated as a result, not needing any additional computation. However, a simple abstraction such as $\lambda x. x$ is a value of $\lambda_c$, and could be translated as such in $\lambda_{alloc}$, but is not a result of $\lambda_{alloc}$. The correct translation is rather the configuration $\{ l \mapsto \lambda x. x \} \vdash l$. The drawback of such a method is that the translation is no longer compositional, at least in the usual sense. Indeed, the translation of an application such as $(\lambda x. x)(\lambda x. x)$ is not the application of the translation of the function to the translation of the argument.

Definition

In order to overcome this difficulty, we introduce a non-standard notion of contexts in $\lambda_{alloc}$, which take as an argument configurations, rather than just expressions. Configurations are pairs of a heap and a multiple lift context, and the application of a context $\Theta \vdash \Phi$ to a configuration $\Theta' \vdash E$ is $\Theta + \Theta' \vdash \Phi[E]$.

We are not done yet. We have indeed seen that results in $\lambda_c$ can be of the shape $b_v \vdash v$. We imagine that $b_v$ will be translated as the heap, roughly. But heaps of $\lambda_{alloc}$ only contain heap blocks, i.e. dummy blocks, functions or evaluated records. Therefore, in the case where $b_v$ contains definitions of the shape $x = y$ for example (or $x = 1$ if we had constants), we have to find another solution. Furthermore, this solution has to take into account the asymmetry of let rec in $\lambda_c$. Indeed, the heap $x = y, z = x$ in fact maps both $x$ and $z$ to the value $y$. Our solution is to retain the part of $\lambda_c$ heaps that cannot be included in $\lambda_{alloc}$ heaps as substitutions. For instance, the $\lambda_c$ binding $x = y, z = x$ is translated as the substitution $\{ z \mapsto x \} \circ \{ x \mapsto y \}$ (recall that composition of substitution is “left to right”).

But then, contexts again become a bit more complicated, because they must include a substitution part. Indeed, the $\lambda_c$ context $x = y, z = x \vdash \square$ does not correspond to any standard evaluation context in $\lambda_{alloc}$. Instead, we have to define a stronger kind of evaluation contexts, including a heap $\Theta$, a standard context $\Phi$, and a substitution $\sigma$. We write them $\Theta \vdash \Phi[\sigma]$, and denote them by $\Psi$.

Applying a context to a configuration is valid if the two heaps define disjoint sets of locations, and if the substitution carried by the context is correct for the configuration, in the following sense.

Definition 20 (Substitution correctness) A substitution $\sigma$ is correct for an expression $E$ iff

$$\forall x \in \text{dom}(\sigma), \sigma(x) \notin \text{Capt}_x(E).$$

This definition extends straightforwardly to heaps and configurations. Fortunately, when the proposed substitution is not correct for the considered configuration, structural equivalence allows to rename all the problematic binders in it, and find an equivalent configuration for which the substitution is correct.

Similarly, the composition $\Psi_1 \circ \Psi_2$ of two contexts $\Psi_i = \Theta_i \vdash \Phi_i[\sigma_i]$ is $\Theta_1 + \Theta_2 \vdash \Phi_1[\Phi_2[\sigma_2 \circ \sigma_1]]$, provided the substitution $\sigma_2 \circ \sigma_1$ is correct for the heap $\Theta_1 + \Theta_2$ and the context $\Phi_1[\Phi_2]$. But again, structural equivalence always allows to find correct equivalent contexts (since binders in contexts are not in position to capture the placeholder).
Properties

In this section, we prove some properties of stability of the reduction relation inside contexts. Not every reduction step is valid inside contexts, since for instance the \textsc{Let} and \textsc{EmptyLet} are only valid at toplevel. However, we will see that inside contexts of the shape $\Theta \vdash \square[\sigma]$, reduction is preserved.

We first prove that contraction is preserved under correct substitution.

**Proposition 11** If $C_1 \rightsquigarrow C_2$ and $\sigma$ is correct for $C_1$ and $C_2$, then $C_1[\sigma] \rightsquigarrow C_2[\sigma]$.

**Proof** By case on the applied contraction rule. Let $C_i = \Theta_i \vdash E_i$, for $i = 1, 2$.

**Beta.** Then $E_1 = lV$, and $\Theta_1 = \Theta_2$, and $\Theta_1(l) = \lambda x. E$. We have $E_1[\sigma] = l(V[\sigma])$, and as $\sigma$ is correct, $(\lambda x. E)[\sigma] = \lambda x. (E[\sigma])$. So $\Theta_1[\sigma] \vdash l(V[\sigma]) \rightsquigarrow E[\sigma][x \mapsto (V[\sigma])]$. As $\sigma$ is correct for $C_1$, $x$ is not in the domain or codomain of $\sigma$, so $\sigma \circ \{ x \mapsto (V[\sigma]) \} = \{ x \mapsto V \} \circ \sigma$, and therefore $C_1[\sigma] \rightsquigarrow \Theta_2[\sigma] \vdash E_2[\sigma]$.

**Allocate, Update, Project.** Similar.

**Lift.** We again have $\Theta_1 = \Theta_2$, with $E_1 = \Lambda[\text{let } B \text{ in } E]$ and $E_2 = \text{let } B \text{ in } \Lambda[E]$. By the side condition on the \textsc{Lift} rule, we also know that $\text{dom}(B) \perp \text{FV}(\Lambda)$. By hypothesis, we finally have $\text{dom}(B)$ disjoint from the domain and codomain of $\sigma$.

So, $C_1[\sigma] = \Theta_1[\sigma] \vdash \Lambda[\sigma][\text{let } B[\sigma] \text{ in } E[\sigma]]$, which reduces to $\Theta_1[\sigma] \vdash \text{let } B[\sigma] \text{ in } \Lambda[\sigma][E[\sigma]]$, as expected.

\[\square\]

This property extends to computational reduction.

**Proposition 12** If $C_1 \longrightarrow C_2$ and $\sigma$ is correct for $C_1$ and $C_2$, then $C_1[\sigma] \longrightarrow C_2[\sigma]$.

**Proof** By case on the applied rule. Let again $C_i = \Theta_i \vdash E_i$, for $i = 1, 2$.

**Context.** By application of the previous proposition.

**EmptyLet.** Trivial.

**Let.** We have $C_1 = \Theta_1 \vdash x = V, B \text{ in } E$, and $C_2 = \Theta_1 \vdash \text{let } B \{ x \mapsto V \} \text{ in } E \{ x \mapsto V \}$. So,

$$C_1[\sigma] = \Theta_1[\sigma] \vdash x = (V[\sigma]), B[\sigma] \text{ in } (E[\sigma]),$$

which reduces to

$$\Theta_1[\sigma] \vdash \text{let } B[\sigma][x \mapsto (V[\sigma])] \text{ in } (E[\sigma][x \mapsto (V[\sigma])]),$$

but as $x$ is not in the domain or codomain of $\sigma$, the substitution $\sigma \circ \{ x \mapsto (V[\sigma]) \}$ is equal to $\{ x \mapsto V \} \circ \sigma$, so $C_1[\sigma]$ reduces to

$$\Theta_1[\sigma] \vdash \text{let } B\{ x \mapsto V \circ \sigma \} \text{ in } (E \{ x \mapsto V \circ \sigma \}),$$

which is exactly $C_2[\sigma]$.

\[\square\]

Now, we prove that reduction by the \textsc{Context} rule is preserved inside any evaluation context.
Evaluation context:
\[ \Psi ::= \Theta \vdash \Phi [\sigma] \]

Restricted evaluation context:
\[ \phi ::= \Theta \vdash \Box [\sigma] \]

Figure 7.17: Evaluation contexts in \( \lambda_{alloc} \)

**Proposition 13** If \( C_1 \xrightarrow{\text{CONTEXT}} C_2 \), then for any context \( \Psi, \Psi[C_1] \xrightarrow{\text{CONTEXT}} \Psi[C_2] \).

**Proof** Let \( C_1 = \Theta_1 \vdash E_1, C_2 = \Theta_2 \vdash E_2, C'_1 = \Psi[C_1], C'_2 = \Psi[C_2], \) and \( \Psi = \Theta \vdash \Phi [\sigma] \). Let us assume w.l.o.g. that \( \sigma \) is correct for the considered objects. Then, \( C'_1 = (\Theta_1 + \Theta)\{\sigma\} \vdash \Phi[E_1]\{\sigma\} \) and \( C'_2 = (\Theta_2 + \Theta)\{\sigma\} \vdash \Phi[E_2]\{\sigma\} \).

Let us prove first that \( C''_1 \xrightarrow{\text{CONTEXT}} C''_2 \), with \( C''_1 = (\Theta_1 + \Theta) \vdash \Phi[E_1] \) and \( C''_2 = (\Theta_2 + \Theta) \vdash \Phi[E_2] \).
As we know, \( C_1 \) reduces to \( C_2 \) by rule CONTEXT, so in fact, \( E_1 = \Phi[E'_1], E_2 = \Phi[E'_2] \), and the proof of \( C_1 \xrightarrow{\text{---}} C_2 \) is of the shape:

\[
\frac{\Theta_1 \vdash E'_1 \rightarrow_e \Theta_2 \vdash E'_2}{C_1 \xrightarrow{\text{---}} C_2}
\]

But it is trivial that contraction rules are not affected by additional bindings in the heap, so we obtain easily that

\[
\Theta + \Theta_1 \vdash E'_1 \rightarrow_e \Theta + \Theta_2 \vdash E'_2
\]

Then, by rule CONTEXT, we have

\[
C''_1 \xrightarrow{\text{---}} C''_2.
\]

Finally, by proposition 12, we deduce that

\[
C''_1\{\sigma\} \xrightarrow{\text{---}} C''_2\{\sigma\},
\]

which is the expected result.

\[\square\]

Now, we would like a similar property to be true with any reduction, but we have seen that it does not hold because of the toplevel nature of the LETREC rule. However, we have a slightly property, with contexts of the shape \( \Theta \vdash \Box [\sigma] \), which we denote by the meta-variable \( \phi \), and call weak evaluation contexts. (The two notions of contexts introduced in this section are recalled in figure 7.17.) A toplevel reduction remains toplevel inside a weak evaluation context.

**Proposition 14** If \( C_1 \xrightarrow{\text{---}} C_2 \), then \( \phi[C_1] \xrightarrow{\text{---}} \phi[C_2] \).

### 7.4.2 Definition of the two translations

This section describes the translation. It consists in fact in two translations. The first one, called the standard translation, is very intuitive, but not easily proved correct. The second one is much less intuitive, but is easier to prove correct. The key technical point is that the standard translation
Translation of expressions:

\[
\begin{align*}
\llbracket x \rrbracket & = x \\
\llbracket \lambda x.e \rrbracket & = \lambda x.\llbracket e \rrbracket \\
\llbracket e_1 \cdot e_2 \rrbracket & = \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\
\{\ldots X_i = e_i \ldots \} & = \{\ldots X_i = \llbracket e_i \rrbracket \ldots \} \\
\llbracket e \cdot X \rrbracket & = \llbracket e \rrbracket \cdot X \\
\llbracket \text{let rec } b \text{ in } e \rrbracket & = \text{let } \text{Dummy}(b), \text{Update}(b) \text{ in } \llbracket e \rrbracket
\end{align*}
\]

Dummy pre-allocation of bindings:

\[
\begin{align*}
\text{Dummy}(\epsilon) & = \epsilon \\
\text{Dummy}(x = e, b) & = (x = \text{alloc } n, \text{Dummy}(b)) \text{ if Size}(e) = n \\
\text{Dummy}(x = e, b) & = \text{Dummy}(b) \text{ if Size}(e) = \epsilon
\end{align*}
\]

Computation of bindings:

\[
\begin{align*}
\text{Update}(\epsilon) & = \epsilon \\
\text{Update}(x = e, b) & = (y = (\text{update } x [e]), \text{Update}(b)) \text{ if Size}(e) = n, \text{ with } y \text{ fresh} \\
\text{Update}(x = e, b) & = (x = [e], \text{Update}(b)) \text{ if Size}(e) = \epsilon
\end{align*}
\]

Figure 7.18: Translation (standard translation)

reduces to the second translation, without using the Beta or Project rules, and therefore without performing any real computation.

Both translations rely on a function Size from to \(\lambda_e\) expressions to \(\mathbb{N} \cup \{\varepsilon\}\). This function is supposed to guess the size of the result of the translation of its argument. We assume that the size of any expression of predictable shape is known, and moreover that the size of variables is undefined. In other words, for any \(e_i \in \text{Predictable}, \text{Size}(e_i) \neq \varepsilon\), and for any variable \(x\), Size\((x) = \varepsilon\).

The standard translation The standard translation is defined in figure 7.18. It is almost direct for variables, functions, applications, and record operations, but the translation of bindings is more intricate. The translation of a binding \(b\) is the concatenation of two bindings in \(\lambda_{alloc}\). The first of them is called the pre-allocation binding, and gives instructions to allocate dummy blocks on the heap for definitions of known size. The second binding is called the update binding. It computes definitions, and alternatively updates the previously pre-allocated dummy blocks for definitions of known sizes, or simply binds the result for definitions of unknown sizes. As announced, this translation does not map results to results. A simple example is \(\lambda x.x\), which is translated as \(\lambda x.x\). To reach a result, this translation still has to reduce to the configuration \(\{l \mapsto (\lambda x.x)\} \vdash l\).

The second translation, named the TOP translation, performs all this kind of reductions at the meta-level, in order to associate results to results. As a consequence, it associates \(\lambda_{alloc}\) configurations to \(\lambda_e\) expressions, and \(\lambda_{alloc}\) configurations to \(\lambda_e\) configurations. It is defined in figures 7.19 and 7.20.

The TOP translation The idea is that the TOP translation is used until the current point of evaluation in the expression, and beyond that point, the standard translation is used.

Variables are still translated as variables. A function \(\lambda x.e\) is translated as with the standard translation, i.e. \(\lambda x.[e]\), but the result is allocated on the heap, at a fresh location \(l\): \(\{l \mapsto \lambda x.[e]\} \vdash l\).
Translation of expressions as configurations:

\[ [x]^{\text{TOP}} = \emptyset \vdash x \]
\[ [\lambda x.e]^{\text{TOP}} = \{l \mapsto \lambda x.\llbracket e \rrbracket\} \vdash l \]
\[ \llbracket \{s_v\} \rrbracket^{\text{TOP}} = \Theta + \{l \mapsto \{s_v\}\} \vdash l \quad \text{for } \llbracket s_v \rrbracket^{\text{TOP}} = \Theta \vdash S_v \]
\[ \llbracket \{s_v, X = e, s\} \rrbracket^{\text{TOP}} = \Theta_1 + \Theta_2 + \{s_v, X = E, \llbracket s \rrbracket\} \quad \text{for } \llbracket s_v \rrbracket^{\text{TOP}} = \Theta_1 \vdash S_v \]
\[ \llbracket ve \rrbracket^{\text{TOP}} = \Theta_1 + \Theta_2 \vdash VE \quad \text{for } \llbracket e \rrbracket^{\text{TOP}} = \Theta_2 \vdash E \]
\[ \llbracket e_1 e_2 \rrbracket^{\text{TOP}} = \Theta \vdash E \llbracket e_2 \rrbracket \quad \text{for } \llbracket e_1 \rrbracket^{\text{TOP}} = \Theta \vdash E \]
\[ \llbracket e.X \rrbracket^{\text{TOP}} = \Theta \vdash E.X \quad \text{for } \llbracket e \rrbracket^{\text{TOP}} = \Theta \vdash E \]
\[ \llbracket \text{let rec } b \text{ in } e \rrbracket^{\text{TOP}} = \begin{cases} \llbracket b \rrbracket^{\text{TOP}} \llbracket \emptyset \vdash e \rrbracket & \text{if } b \text{ is not evaluated} \\ \llbracket b \rrbracket^{\text{TOP}} \llbracket e \rrbracket^{\text{TOP}} & \text{otherwise} \end{cases} \]

Translation of configurations:

\[ \llbracket b \vdash e \rrbracket^{\text{TOP}} = \llbracket \text{let rec } b \text{ in } e \rrbracket^{\text{TOP}} \]

Translation of bindings and evaluated records:

\[ \llbracket b_v, b \rrbracket^{\text{TOP}} = TDun(b) \circ \text{TOP}(b_v) \circ TUp(b) \quad \text{where } b \not\equiv (x = v, b') \]
\[ \llbracket X_1 = v_1 \ldots X_n = v_n \rrbracket^{\text{TOP}} = \bigcup_{1 \leq i \leq n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n) \quad \text{with } \forall i, \llbracket \alpha_i \rrbracket^{\text{TOP}} = \Theta_i \vdash V_i \]

Figure 7.19: The TOP translation (first part)
Translation of evaluated bindings: Ev. binding → (heap × substitution × variable allocation)

\[
\begin{align*}
\TOP(\varepsilon) &= \emptyset \vdash (id, id) \\
\TOP(x = v, b_v) &= \Theta \vdash (\sigma \circ \{x \mapsto V\}, \eta) \quad \text{if} \quad \begin{cases} 
  \text{Size}(v) = [?] \\
  [v]^{\TOP} = \emptyset \vdash V \\
  \TOP(b_v) = \Theta \vdash (\sigma, \eta)
\end{cases} \\
\TOP(x = v, b_v) &= \Theta \vdash (\sigma, \eta \cup \{x \mapsto l\}) \quad \text{if} \quad \begin{cases} 
  \text{Size}(v) = n \\
  [v]^{\TOP} = \Theta \vdash l \\
  \TOP(b_v) = \Theta \vdash (\sigma, \eta)
\end{cases}
\end{align*}
\]

Actual dummy pre-allocation: Binding → (heap × variable allocation)

\[
\begin{align*}
\TDum(\varepsilon) &= \emptyset \vdash id \\
\TDum(x = e, b) &= \TDum(b) \quad \text{if} \quad \text{Size}(v) = [?]
\end{align*}
\]

Actual computation of bindings: Binding → (heap × binding of \(\lambda_{\text{alloc}}\))

\[
\begin{align*}
\TUpsilon(\varepsilon) &= \emptyset \vdash \varepsilon \\
\TUpsilon(x = e, b) &= \Theta_1 + \Theta_2 \vdash x = E, B \quad \text{if} \quad \begin{cases} 
  \text{Size}(v) = [?] \\
  [e]^{\TOP} = \Theta_1 \vdash E \\
  \TUpsilon(b) = \Theta_2 \vdash B
\end{cases} \\
\TUpsilon(x = e, b) &= \Theta_1 + \Theta_2 \vdash y = (\text{update} \ xE), B \quad \text{if} \quad \begin{cases} 
  \text{Size}(v) = n \\
  [e]^{\TOP} = \Theta_1 \vdash E \\
  \TUpsilon(b) = \Theta_2 \vdash B \\
  y \text{ fresh}
\end{cases}
\end{align*}
\]

Figure 7.20: The TOP translation (continued): bindings
An evaluated record takes the translations of its fields and puts them in a record allocated on the
heap at a fresh location \( l \): \( \Theta \vdash \{ l \rightarrow \{ s_v \} \} \vdash l \). Here, \( \Theta \vdash s_v \) is the translation of the record \( s_v \),
defined in figure 7.19. If \( s_v = (X_1 = v_1 \ldots X_n = v_n) \), and for each \( i \), \( [v_i]^{\top} = \Theta_i \vdash V_i \), then
\( \Theta \vdash s_v = \bigcup_{1 \leq i \leq n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n) \).

When the record is not fully evaluated, it is not yet allocated on the heap. It is divided into its
evaluated part \( s_v \), and the rest \( X = e.s \). \( s_v \) is translated as for evaluated records, into \( \Theta_1 \vdash s_v \).
The field \( e \) is translated with the \( \top \) translation, into \( \Theta_2 \vdash E \), and \( s \) is translated with the
standard translation. We denote by \( [s] \) the record \( s \), translated with the standard translation.

Function application works like records: if the function part is not a value, then it is translated
with the \( \top \) translation, while the argument is translated with the standard translation. If the
function is a value, then both parts are translated with the \( \top \) translation.

The translation of a record selection \( e.X \) consists in translating \( e \) with the \( \top \) translation, and
then selecting the field \( X \).

**TOP translation of bindings** The translation of bindings is more complicated. As for records,
the binding is divided into its evaluated part \( b_v \) and the rest \( b \), which can be empty, but does not
begin with a value.

The rest of the binding \( b \) is translated as follows. The pre-allocation pass, in the standard
translation, consists in giving instructions for allocating dummy blocks. Here, these blocks are directly
allocated by the function \( TDum \), which returns the heap of dummy blocks, and the substitution
replacing variables with the corresponding locations. The update pass, in the standard translation,
consists in either updating a dummy block with the translation of the definition, or simply
binding it. Here, it is almost the same, except that the first definition is translated with the \( \top \) translation,
while the remaining ones are translated with the standard translation. The \( TUp \) is in charge of these operations.

Roughly, the binding \( b_v \) is translated as a heap and a substitution, by the \( \top \) function. Definitions
of unknown size \( x = v \) yield a translation of the shape \( \emptyset \vdash V \), and are included in the translation
as a substitution \( x \mapsto V \). Definitions of known size \( x = v \) are translated as a heap and a variable
allocation: \( v \) has a translation of the shape \( \Theta \vdash l \), and it is included in the translation of \( b_v \) as \( \Theta \),
and the allocation \( x \mapsto l \).

In practice, it is useful to distinguish substitutions coming from definitions of unknown size, which
can be of any shape, from substitutions coming from definitions of known size, which are allocations,
and therefore have the shape \( x \mapsto l \). Indeed, when putting the results together, it is important
to take the order into account, for definitions of unknown size. For instance, a binding such as
\( y = z, x = y \) generates two substitutions \( y \mapsto z \) and \( x \mapsto y \), but the first one must be performed last.
This is why, according to the definition of \( \top \), the result would be \( \{ x \mapsto y \} \circ \{ y \mapsto z \} \). This works
because syntactically, definitions of unknown size can only be mentioned by subsequent definitions
in the binding. However, definitions of known size can be mentioned by previous definitions. The
key is that the substitutions they generate are allocations, so they are not modified by other
substitutions, and can be performed right in the end. Formally, the translation of \( b_v \) is a heap \( \Theta \),
a substitution \( \sigma \), corresponding to the definitions of unknown size, and an allocation \( \eta \), giving
the locations allocated in \( \Theta \) for the definitions of known size. Semantically, it corresponds to a heap
\( \Theta \) and the substitution \( \sigma \circ \eta \), and will be used as such.

The three functions for translating bindings, \( TDum \), \( TUp \), and \( \top \), can be viewed as contexts.
The \( TDum \) returns a heap \( \Theta \) and an allocation \( \eta \), and it forms a context \( \Theta \vdash \square[\eta] \). The \( TUp \)
function returns a heap \( \Theta \) and a binding \( B \), which form a context \( \Theta \vdash \{ \top \vdash \} \vdash B \) in \( \square[\eta] \). The
\( \top \) function returns a heap \( \Theta \), a substitution \( \sigma \), and an allocation \( \eta \), and it forms a context
\( \Theta \vdash \square[\sigma \circ \eta] \). Notice that the context corresponding to \( TUp \) is not an evaluation context. In
case the whole binding \( b_v, b \) is evaluated (i.e. \( b \) is empty), then the contexts for pre-allocation
and update, $TDum(b)$ and $TU_p(b)$ are empty, and the translation of let rec $b; e$ in $e$ is the TOP translation of $e$, $[e]^{TOP}$, put in the context $TOP(b_v)$. Otherwise, the translation of let rec $b; e$ in $e$ is the standard translation of $e$, put in the context $TDum(b) \circ TOP(b_v) \circ TU_p(b)$.

### 7.4.3 Relating the two translations

An interesting fact is that the standard translation of any expression reduces to its TOP translation, in any context. The proof of this property is in three steps. First, we prove it for values. Then, we prove that the standard translation of a binding reduces to its TOP translation. Finally, we prove the expected result.

In fact, for values, we prove a more powerful result, namely that the standard translation reduces to the TOP translation, but only by rule CONTEXT, with a premise using ALLOCATE, which we write CONTEXT (ALLOCATE).

We make some additional hypotheses related to the correctness of the Size function.

**Hypothesis 3** For all expressions $e, f, e'$, for all value $v$, for all bindings $b, b'$, for all substitution $\sigma$, for all context $C$:

- If $\text{Size}(e) = n$ and $b \vdash e \rightarrow b' \vdash e'$, then $\text{Size}(e') = n$;
- If $\text{Size}(v) = n$, then there exist $\Theta$ and $l$ such that $[v]^{TOP} = \Theta \vdash l$ and $\text{Size}(\Theta(l)) = n$;
- If $\text{Size}(e) = \text{Size}(f) = n$, then $\text{Size}(C[e]) = \text{Size}(C[f])$.
- $\text{Size}(e\{\sigma\}) = \text{Size}(e)$;
- $\text{Size}(\text{let rec } b \text{ in } e) = \text{Size}(e)$.

**Proposition 15 (Translation of values reduces to TOP)** For all context $\Psi$ and for all value $v$, $\Psi[\emptyset \vdash [v]] \xrightarrow{*} \Psi[[v]^{TOP}]$, only by rule CONTEXT (ALLOCATE).

**Proof** By induction on $v$.

- $v = x$, trivial.
- $v = \lambda x.e$. Then $[v] = \lambda x.[e]$, so in any context $\emptyset \vdash [v]$ reduces in one CONTEXT (ALLOCATE) step to $\{l \mapsto \lambda x.[e]\} \vdash l$, which is the TOP translation of $v$.
- $v = \{X_1 = v_1 \ldots X_n = v_n\}$. By induction hypothesis, for any context $\Psi_i$, for each $i$, we have
  $$\Psi_i[\emptyset \vdash [v_i]] \xrightarrow{*} \Psi_i[[v_i]^{TOP}]$$
  Let for each $i$, $[v_i]^{TOP} = \Theta_i \vdash V_i$. By a trivial induction on $n$, we prove that for any context $\Psi$,
  $$\Psi[\emptyset \vdash [[X_1 = v_1 \ldots X_n = v_n]]] \xrightarrow{*} \Psi[\biguplus_{1 \leq i \leq n} \Theta_i \vdash \{X_1 = V_1 \ldots X_n = V_n\}]$$
  only by rule CONTEXT (ALLOCATE). By proposition 13, this configuration in turn reduces by rule CONTEXT (ALLOCATE) to
  $$\Psi[\biguplus_{1 \leq i \leq n} \Theta_i + \{l \mapsto \{X_1 = V_1 \ldots X_n = V_n\}\} + l],$$
  which is exactly $[v]^{TOP}$.

$\square$
Corollary 9 For all weak evaluation context $\phi$, expression $E$, and binding $b$ of the shape $b = (x = v, b')$,

$$\phi \circ \text{Update}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \text{TOP}(b)[\emptyset \vdash E]$$

Proof We know that $\phi \circ \text{Update}(b)[\emptyset \vdash E] = \phi[\text{let } y = \Phi[[v]], \text{Update}(b') \text{ in } E]$, where

$$(y, \Phi) = \begin{cases} (x, \Box) & \text{if Size}(v) = [?] \\ (z, \text{update } x \Box) & \text{otherwise} (z \text{ fresh}). \end{cases}$$

This expression can be seen as $\Psi[\emptyset \vdash v]$ for some $\Psi$. By proposition 15, it reduces to $\Psi[[v]]^\text{TOP}$, so we obtain $\phi \circ \text{let } y = \Phi, \text{Update}(b') \text{ in } E[[v]]^\text{TOP}$, which is exactly $\phi \circ \text{TOP}(b)[\emptyset \vdash E]$. □

Now, let us have a look at the translation of bindings. The TOP translation splits the bindings in two, according to the first non-value definition. But of course, one could split at another point, provided the first part contains only values. Indeed, the first part is given as an argument to the $\text{TOP}$ function, which is defined only on evaluated bindings, whereas the second part is given as an argument to the $\text{TDum}$ and $\text{TU}_{p}$ functions, which work as well on value and non-value definitions. We call a partial translation of a binding $b = b_{v}, b_{v}', b'$ its TOP translation, computed as if $b_{v}'$ was not evaluated, i.e., $\text{TDum}(b_{v}, b', b') \circ \text{TOP}(b_{v}) \circ \text{TU}_{p}(b_{v}', b')$. We prove that any partial translation reduces to the TOP translation. We proceed in three main steps: first, we prove that the pre-allocation pass is performed at the object level by the code generated by the $\text{Dummy}$ function, and at the meta level by the $\text{TDum}$ function, in the same way; then we prove a similar property for the functions $\text{Update}$ and $\text{TU}_{p}$; and we eventually connect the two to prove the whole desired property.

Proposition 16 (Dummy) For all binding $B$, for all weak evaluation context $\phi$,

$$\phi[\emptyset \vdash \text{Dummy}(b), B \text{ in } E] \longrightarrow^* (\phi \circ \text{TDum}(b))[\emptyset \vdash \text{let } B \text{ in } E].$$

Proof By induction on $b$. If $b$ is empty, then there is nothing to prove. Otherwise, we are in one of the following cases.

- $b = (x = e, b')$, with $\text{Size}(e) = [?]$. Then $\text{Dummy}(b) = \text{Dummy}(b')$ and $\text{TDum}(b) = \text{TDum}(b')$, so by induction hypothesis, we obtain the expected result.

- $b = (x = e, b')$, with $\text{Size}(e) = n$. Then $\text{Dummy}(b) = (x = \text{alloc } n, \text{Dummy}(b'))$. Let $\text{TDum}(b') = \Theta + \eta$, we have $\text{TDum}(b) = \Theta + \{l \mapsto \text{alloc } n\} + \eta \cup \{x \mapsto l\}$, for a fresh $l$. Let $\phi$ be a weak evaluation context, and $E_{G} = \phi[\text{let } \text{Dummy}(b), B \text{ in } E]$. We have $E_{G} \rightarrow \phi[\{l \mapsto \text{alloc } n\} \vdash \{x \mapsto l\}]$. By rule CONTEXT (ALLOCATE), we have $E_{G} \rightarrow \phi[\{l \mapsto \text{alloc } n\} \vdash \text{let } x = l, \text{Dummy}(b'), B \text{ in } E]$. By proposition 14, this last expression reduces to $\phi[\{l \mapsto \text{alloc } n\} \vdash \{\text{let } \text{Dummy}(b'), B \text{ in } E\}[x \mapsto l]]$. Let $\phi_{e} = \text{TDum}(x = e) = \{l \mapsto \text{alloc } n\} \vdash \Box[\{x \mapsto l\}]$ and $\phi_{e} = \phi \circ \phi_{e}$; we can view the expression as $\phi_{e}[\emptyset \vdash \text{Dummy}(b'), B \text{ in } E]$, which by induction hypothesis reduces to $\phi_{e}[\text{TDum}(b')[\emptyset \vdash \text{let } B \text{ in } E]]$. In other words, we obtain $\phi[\text{TDum}(x = e) \circ \text{TDum}(b')[\emptyset \vdash \text{let } B \text{ in } E]]$, which is the expected result, since obviously $\text{TDum}(x = e) \circ \text{TDum}(b') = \text{TDum}(b)$.

□

Proposition 17 (Update) Let $b = (x = v, b')$. For all weak evaluation context $\phi$, for all expression $E$, we have

$$\phi \circ \text{TDum}(b) \circ \text{TU}_{p}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \text{TDum}(b') \circ \text{TOP}(x = v) \circ \text{Update}(b')[\emptyset \vdash E].$$

Proof
• If Size(v) = n, then \[v\]^{TOP} = \Theta_v \vdash l, and we have

\[TU_P(b) = \Theta_v \vdash y = update x l, Update(b'),\]

with a fresh y. Alternatively, we can choose another fresh location l' for the result, and have

\[[v]^{TOP} = \Theta'_v \vdash l', with \Theta'_v = \Theta_v \cup \{l' \mapsto \Theta_v(l)\}.

Let \(E_\emptyset = \phi \circ TDum(b) \circ TU_P(b)[\emptyset \vdash E].\)

We have \(E_\emptyset = \phi \circ TDum(b)[\Theta'_v \vdash \text{let } y = update x l', Update(b') \in E], and also Size(v) = n\) and \(TDum(b) = TDum(b') \circ \{l \mapsto \text{alloc } n\} \vdash \{x \mapsto l\}.\) So

\[E_\emptyset = \phi \circ TDum(b')[\Theta'_v + \{l \mapsto \text{alloc } n\} \vdash \text{let } y = update x l', Update(b') \in E}\{x \mapsto l\}].\]

But by hypothesis 3, Size(\(\Theta'_v(l')\)) = n, so rule Update applies, and \(E_\emptyset\) reduces to

\[\phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\}) \vdash \text{let } y = \{\}, \text{Update}(b') \in E]\{x \mapsto l\}],\]

and then, as y is fresh, by rule LET to

\[\phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\}) \vdash \text{let } Update(b') \in E]\{x \mapsto l\}].\]

But the location l' is not used anymore, so by rule GC, the obtained expression reduces to

\[\phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\}) \vdash \text{let } Update(b') \in E]\{x \mapsto l\}].\]

And finally, we notice that \(\Theta'_v(l') = \Theta_v\), so \(E_\emptyset\) reduces to

\[\phi \circ TDum(b')[(\Theta_v \vdash \text{let } Update(b') \in E]\{x \mapsto l\}]] = \phi \circ TDum(b') \circ TOP(x = v)\{x \mapsto l\}]] = \phi \circ TDum(b') \circ TOP(x = v) \circ Update(b')[\emptyset \vdash E].\]

• If Size(v) = \([?]\), then there exists a y such that \[[v]^{TOP} = \emptyset \vdash y, so

\[TU_P(b) = \emptyset \vdash x = y, Update(b').\]

Let \(E_\emptyset = \phi \circ TDum(b) \circ TU_P(b)[\emptyset \vdash E].\)

We have \(E_\emptyset = \phi \circ TDum(b)[\emptyset \vdash \text{let } x = y, Update(b') \in E]\), and by rule LET, by proposition 14, \(E_\emptyset \rightarrow \phi \circ TDum(b)[\emptyset \vdash \{\text{let } Update(b') \in E}\{x \mapsto y\}]\).

But \(TOP(x = v) = TOP(x = y) = \emptyset \vdash \{x \mapsto y, id\}, so E_\emptyset \rightarrow \phi \circ TDum(b) \circ TOP(x = v)\{\emptyset \vdash let Update(b') \in E], which is the expected result.

\[\square\]

**Proposition 18 (Pre-allocated locations are definitive)** If \(TDum(b_v) = \Theta_1 \vdash \eta_1\), then there exist \(\Theta_2, \sigma_2, \eta_2\) such that \(TOP(b_v) = \Theta_2 \vdash (\sigma_2, \eta_2)\) and \(\eta_1 = \eta_2\).

In the following proposition, we consider a substitution \(\sigma\) as a context \(\emptyset \vdash \Box[\sigma]\).

**Proposition 19 (Decomposition of the translation of evaluated bindings)** Let \(b_v = (x = v, b_v')\) and \(TDum(b_v') = \Theta_{b_v'} \vdash \eta_{b_v'}\). We have

\[TOP(b_v) = \eta_{b_v'} \circ TOP(x = v) \circ TOP(b_v').\]

179
Proof Let \( \text{TOP}(x = v) = \Theta_v \vdash (\sigma_v, \eta_v) \), and \( \text{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \). We have \( \text{TOP}(b_v) = \Theta_v + \Theta_{b_v} \vdash (\sigma \circ \sigma_v, \eta \cup \eta_v) \). By proposition 18, we can choose \( \Theta, \sigma, \) and \( \eta \) such that \( \eta = \eta_{b_v} \).

Then,
\[
\eta_{b_v} \circ \text{TOP}(x = v) \circ \text{TOP}(b_v) \\
= \Theta + \Theta_v \vdash \sigma \circ \eta \circ \sigma_v \circ \eta_{b_v} \circ \eta_v \\
= \Theta + \Theta_v \vdash \sigma \circ \eta_{b_v} \circ \sigma_v \circ \eta_v \\
= \Theta + \Theta_v \vdash (\sigma \circ \sigma_v) \circ (\eta_{b_v} \cup \eta_v) \\
= \text{TOP}(b_v)
\]

\( \Box \)

Proposition 20 (Commuting contexts) Let \( \phi_1 = \Theta_1 \vdash \Box[\sigma_1] \) and \( \phi_2 = \Theta_2 \vdash \Box[\sigma_2] \). If \( \text{dom}(\sigma_2) \perp \sigma_1 \) and \( \sigma_1^2 = \sigma_1 \), then \( \phi_1 \circ \phi_2 = \sigma_1 \circ \phi_2 \circ \phi_1 \).

Proof This property is simple, provided \( \sigma_2 \circ \sigma_1 = \sigma_1 \circ \sigma_2 \circ \sigma_1 \). Recall that \( \text{dom}(\sigma_2) \perp \sigma_1 \). We prove that the two total functions \( \sigma = \sigma_2 \circ \sigma_1 \) and \( \sigma' = \sigma_1 \circ \sigma_2 \circ \sigma_1 \) from variables to values are pointwise equal.

- On \( x \in \text{dom}(\sigma_2) \), by hypothesis \( x \notin \text{dom}(\sigma_1) \), so we have \( \sigma'(x) = x[\sigma_1\{\sigma_2\}\{\sigma_1\}] = x[\sigma_2\{\sigma_1\}] = \sigma(x) \).

- On \( x \notin \text{dom}(\sigma_2) \), distinguish the two cases.
  - If \( x \in \text{dom}(\sigma_1) \), then \( \sigma(x) = x[\sigma_2\{\sigma_1\}] = x[\sigma_1] \). But by hypothesis \( \sigma_1(x) \in \text{cod}(\sigma_1) \perp \text{dom}(\sigma_2) \), so \( \sigma'(x) = x[\sigma_1\{\sigma_2\}\{\sigma_1\}] = \sigma_1(x)[\sigma_2\{\sigma_1\}] = \sigma_1(x)[\sigma_1] = x[\sigma_1^2] = x[\sigma_1] = \sigma(x) \).
  - If \( x \notin \text{dom}(\sigma_1) \), then \( \sigma(x) = x = \sigma'(x) \).

\( \Box \)

Corollary 10 Let \( b_v = (b_{v_1}, b_{v_2}) \) be a syntactically correct binding. Let \( \text{TDum}(b_v) = \Theta_2 \vdash \eta_2 \). We have \( \eta_2 \circ \text{TOP}(b_{v_1}) \circ \eta_2 = \eta_2 \circ \text{TOP}(b_{v_1}) \).

Proof Let \( \text{TOP}(b_{v_1}) = \Theta_1 \vdash (\sigma_1, \eta_1) \). By proposition 20, it is enough to prove \( \text{dom}(\sigma_1 \circ \eta_1) \perp \eta_2 \) and \( \eta_2^2 = \eta_2 \). But we have \( \text{dom}(b_{v_1}) = \text{dom}(b_{v_2}) \), so \( \text{dom}(\sigma_1 \circ \eta_1) = \text{dom}(\eta_2) \). Moreover, \( \text{cod}(\eta_2) \) contains only locations, whereas \( \text{dom}(\sigma_1 \circ \eta_1) \) contains only variables, so \( \text{cod}(\sigma_2) \perp \text{dom}(\sigma_1 \circ \eta_1) \). Finally, as all variable allocations, \( \eta_2 \) is idempotent. \( \Box \)

Corollary 11 Let \( b_v = (b_{v_1}, \sigma_2) \) and \( \text{TDum}(b_v) = \Theta_2 \vdash \eta_2 \). We have
\[
\text{TOP}(b_v) = \eta_2 \circ \text{TOP}(b_{v_1}) \circ \text{TOP}(b_{v_2}).
\]

Proof By induction on \( b_{v_1} \).

- \( b_{v_1} = \epsilon \), because \( \eta_2 \) is idempotent.

180
• $b_{v1} = (x \in v, b_{v1}^\prime)$. Let $b_{v}^\prime = b_{v1}^\prime, b_{v2}$, $TDum(b_{v}^\prime) = \Theta_{b_{v}^\prime} \vdash \eta_{b_{v}^\prime}$, and $TDum(b_{v1}^\prime) = \Theta_{b_{v1}^\prime} \vdash \eta_{b_{v1}^\prime}$. By definition of $TDum$, we have $\eta_{b_{v}^\prime} = \eta_{b_{v1}^\prime} \cup \eta_{2}$. Then, we can calculate

$$TOP(b_{v}) = \eta_{b_{v}^\prime} \circ TOP(x = v) \circ TOP(b_{v}^\prime)$$

(by lemma 19)

$$= \eta_{b_{v}^\prime} \circ TOP(x = v) \circ \eta_{2} \circ TOP(b_{v1}^\prime) \circ TOP(b_{v2})$$

(by induction hypothesis)

$$= \eta_{b_{v}^\prime} \cup \eta_{2} \circ TOP(x = v) \circ \eta_{2} \circ TOP(b_{v1}^\prime) \circ TOP(b_{v2})$$

$$= \eta_{b_{v}^\prime} \circ \eta_{2} \circ TOP(x = v) \circ TOP(b_{v1}^\prime) \circ TOP(b_{v2})$$

(by proposition 10)

$$= \eta_{2} \circ \eta_{b_{v}^\prime} \circ TOP(x = v) \circ TOP(b_{v1}^\prime) \circ TOP(b_{v2})$$

(by proposition 19)

□

**Proposition 21 (TOP Update pass)** For all weak evaluation context $\phi$, and configuration $C$,

$$\phi \circ TDum(b_{v}, b) \circ TUP(b_{v}, b)[C] \rightarrow^* \phi \circ TDum(b) \circ TOP(b_{v}) \circ Update(b)[C].$$

**Proof** By induction on $b_{v}$. If $b_{v} = \epsilon$, there is nothing to prove. Otherwise, let $b_{v} = (x \in v, b_{v}^\prime)$. By proposition 17,

$$\phi \circ TDum(b_{v}, b) \circ TUP(b_{v}, b)[C] \rightarrow^* \phi \circ TDum(b_{v}^\prime, b) \circ TOP(x = v) \circ Update(b_{v}^\prime, b)[C].$$

But by corollary 10, this is equal to

$$\phi \circ \eta \circ TOP(x = v) \circ TDum(b_{v}^\prime, b) \circ Update(b_{v}^\prime, b)[C],$$

where $TDum(b_{v}^\prime, b) = \Theta b_{v}^\prime \vdash \eta_{b_{v}^\prime}$.

By induction hypothesis, we know that the obtained expression reduces to

$$\phi \circ \eta \circ TOP(x = v) \circ TDum(b) \circ TOP(b_{v}^\prime) \circ Update(b)[C].$$

But if we let $TDum(b_{v}^\prime) = \Theta_{1} \vdash \eta_{b_{v}^\prime}$ and $TDum(b) = \Theta_{2} \vdash \eta_{2}$, we have $\eta = \eta_{1} \cup \eta_{2}$, so

$$\phi \circ \eta \circ TOP(x = v) \circ TDum(b) \circ TOP(b_{v}^\prime) \circ Update(b)[C]$$

$$= \phi \circ \eta_{1} \circ \eta_{2} \circ TOP(x = v) \circ TDum(b) \circ TOP(b_{v}^\prime) \circ Update(b)[C]$$

$$= \phi \circ \eta_{1} \circ TDum(b) \circ TOP(x = v) \circ TOP(b_{v}^\prime) \circ Update(b)[C]$$

(by corollary 10)

$$= \phi \circ TDum(b) \circ \eta_{1} \circ TOP(x = v) \circ TOP(b_{v}^\prime) \circ Update(b)[C]$$

(because $TDum(b)$ is not modified by any substitution)

$$\phi \circ TDum(b) \circ TOP(b) \circ Update(b)[C]$$

(by proposition 19)

□

**Proposition 22 (Update pass)** For all weak evaluation context $\phi$, and configuration $C$,

$$\phi \circ TDum(b_{v}, b) \circ Update(b_{v}, b)[C] \rightarrow^* \phi \circ TDum(b) \circ TOP(b_{v}) \circ Update(b)[C].$$

**Proof** By corollary 9, we have

$$\phi \circ TDum(b_{v}, b) \circ Update(b_{v}, b)[C]$$

$$\rightarrow^* \phi \circ TDum(b_{v}, b) \circ TUP(b_{v}, b)[C].$$

By proposition 21, it further reduces to $\phi \circ TDum(b) \circ TOP(b_{v}) \circ Update(b)[C]$. □
Proposition 23 (Partial translation of bindings) For all evaluation context $\Psi$,
\[ \Psi[\emptyset \vdash [\text{let rec } b_v, b \text{ in } e]] \longrightarrow^* \Psi \circ \text{TDum}(b) \circ \text{TOP}(b_v) \circ \text{Update}(b)[\emptyset \vdash e]. \]

Proof Let $\Psi = \Theta \vdash \Phi[\sigma]$, and $\phi = \Theta \vdash \Box[\sigma]$. Let
\[ E_0 = \Psi[\emptyset \vdash [e]] = \Psi[\emptyset \vdash \text{Dummy}(b_v, b), \text{Update}(b_v, b) \text{ in } [e]] \]

By rule Lift and modulo variable renaming, we have
\[ E_0 \longrightarrow^* \phi[\emptyset \vdash \text{Dummy}(b_v, b), \text{Update}(b_v, b) \text{ in } \Phi[[e]]]. \]

By proposition 16, this expression reduces to $\phi \circ \text{TDum}(b_v, b)[\emptyset \vdash \text{Update}(b_v, b) \text{ in } \Phi[[e]]]$. By proposition 22, it in turn reduces to $\phi \circ \text{TDum}(b_v, b) \circ \text{TOP}(b_v) \circ \text{Update}(b)[\Phi[[e]]]$, which is equal to $\Psi \circ \text{TDum}(b_v, b) \circ \text{TOP}(b_v) \circ \text{Update}(b)[[e]]$: $\Box$.

Lemma 39 (Standard translation reduces to TOP translation) For all context $\Psi$ and for all expression $e$,
\[ \Psi[\emptyset \vdash [e]] \longrightarrow^* \Psi[[e]]^{\text{TOP}}. \]

Proof By induction on $e$. If $e$ is a value, we use proposition 15.

Application. Let $e = e_1 e_2$, $\Psi$ be a context, and $E_0 = \Psi[\emptyset \vdash [e]] = \Psi[\emptyset \vdash [e_1][e_2]]$. Let also $[e_1]^{\text{TOP}} = \Theta_1 \vdash E_1$. By induction hypothesis, $E_0 \longrightarrow^* \Psi[\Theta_1 \vdash E_1[e_2]]$. If $e_1$ is not a value, this is directly $\Psi[[e]]^{\text{TOP}}$. Otherwise, $E_1$ is a value, say $V_1$, and $\Psi_0 = \Psi[\Theta_1 \vdash \{V_1 \square\}[id]]$ is an evaluation context, so by induction hypothesis again, if we let $[e_2]^{\text{TOP}} = \Theta_2 \vdash E_2$, then $\Psi_0[\emptyset \vdash [e_2]] \longrightarrow^* \Psi_0[\Theta_2 \vdash E_2]$, which is equal to $\Psi[\Theta_1 + \Theta_2 \vdash V_1 E_2] = \Psi[[e]]^{\text{TOP}}$.

Record field selection. Simple by induction hypothesis.

Record. Let $e = \{s_v, X = f, s\}$, where $f$ is not a value. Let $[s_v]^{\text{TOP}} = \Theta_1 \vdash S_v$. By a trivial induction on $s_v$, we prove that $\Psi[\emptyset \vdash \{\{s_v, X = f, s\}\}] \longrightarrow^* \Psi[\Theta_1 \vdash \{S_v, X = [f], [s]\}]$. This expression can be viewed as $\Psi_0[\emptyset \vdash [f]]$, with $\Psi_0 = \Psi[\Theta_1 \vdash \{S_v, X = \Box, [s]\}]$. Let $[[f]]^{\text{TOP}} = \Theta_2 \vdash F$. By induction hypothesis, the above expression reduces to $\Psi_0[\Theta_2 \vdash F]$, which is equal to $\Psi[\Theta_1 + \Theta_2 \vdash \{S_v, X = F, [s]\}]$, and this is the expected result.

Binding. Let $e = \text{let rec } b \text{ in } f$.

1. If $b = e$, then $[\emptyset]^{\text{TOP}} = \emptyset \vdash \Box[\emptyset id]$, so $[e]^{\text{TOP}} = [[f]]^{\text{TOP}}$. So, $\Psi[\emptyset \vdash [e]] = \Psi[\emptyset \vdash \text{let } \epsilon \text{ in } [f]]$. By rules Lift and then EmptyLet, it reduces to $\Psi[\emptyset \vdash [[f]]]$, which by induction hypothesis reduces to $\Psi[[[f]]^{\text{TOP}}]$, as expected.

2. If $b = b_v$, non empty, then $[e]^{\text{TOP}} = \text{TOP}(b_v)[[[f]]^{\text{TOP}}]$. We have
\[ \Psi[\emptyset \vdash [e]] = \Psi[\emptyset \vdash \text{let rec } b_v \text{ in } [f]] \longrightarrow^* \Psi \circ \text{TOP}(b_v)[\emptyset \vdash [[f]]] \]
(by proposition 23)
\[ \longrightarrow^* \Psi \circ \text{TOP}(b_v)[[[f]]^{\text{TOP}}] \]
(by induction hypothesis)
\[ = \Psi[[e]]^{\text{TOP}} \]
3. If \( b = b_v, b' \), with \( b' \) non empty, then \([e]^{\text{TOP}} = TDum(b') \circ \text{TOP}(b_v) \circ TU_p(b')[\emptyset \vdash [f]]\).

We have

\[
\Psi(\emptyset \vdash [e]) = \Psi(\emptyset \vdash [[\text{let rec } b_v, b' \text{ in } f]])
\]

\[
\rightarrow^* \Psi \circ TDum(b') \circ \text{TOP}(b_v) \circ \text{Update}(b')[\emptyset \vdash [f]]
\]

(by proposition 23)

\[
\rightarrow^* \Psi \circ TDum(b') \circ \text{TOP}(b_v) \circ TU_p(b')[\emptyset \vdash [f]]
\]

(by induction hypothesis)

\[
\Psi[[e]^{\text{TOP}}]
\]

\[\Box\]

7.5 Correctness

7.5.1 Translation of contexts and compositionality

Both the standard and the TOP translations rely on sizes. In a binding, if a definition \( x = e \) is of known size, then it is translated as the binding \( y = \text{update } x \ [e] \), whereas otherwise, it is translated as \( x = [e] \). For this reason, it is not compositional in the usual sense: a straightforward property such as \( [[E \ [e]]] = [[E]] [[e]] \) does not hold. Moreover, there is no straightforward translation for contexts: consider let rec \( x = \square \in \{\} \) for instance; should it be translated as if the expression filling the hole was of known size or unknown size?

The TOP translation retains a kind of compositionality though. We define complete contexts in \( \lambda_c \), as normal contexts, except that the context hole is now annotated with a size indication \( \zeta \in \mathbb{N} \cup \{?\} \). Complete context application is only valid if the argument as the expected size. Complete contexts are then translated exactly as expressions. For this, the definition in figure 7.19 is simply extended with \( [[\square \zeta]]^{\text{TOP}} = [[\square \zeta]] = \square \), given that a context hole \( \square \zeta \) has size \( \zeta \), and that it is not a value. Normal contexts are translated, with an additional argument giving the size of the context hole. For instance, we write \( [[E \ [E]]]^{\text{TOP}} \) for \( [[E \ [\square \zeta]]]^{\text{TOP}} \). The standard translation is compositional for this notion of contexts.

**Proposition 24 (Compositionality of the standard translation)** For all context \( E \) and expression \( e \),

\[
[[E \ [e]]] = [[E \ \text{Size}(e) \ [e]]].
\]

The translation is compositional with respect to this notion of contexts, provided the right size indication is chosen, and that the expression filling the hole is not a value. Indeed, in the translation of bindings, a distinction is made between evaluated and unevaluated definitions, which breaks compositionality in this case, because the context hole is not considered a value. Fortunately, for values, a weaker property of compositionality modulo reduction holds, which allows to prove that the translation is faithfull.

**Proposition 25 (Compositionality for lift contexts)** If \( e \notin \text{Values} \), then

\[
[[L \ [e]]]^{\text{TOP}} = [[L \ \text{Size}(e) \ [e]]^{\text{TOP}}].
\]

**Proof** By case analysis on \( L \). We treat one example case, application: \( L = \square f \). We have \( [[L \ [e]]]^{\text{TOP}} = [[e \ f]]^{\text{TOP}} = \emptyset \vdash [f] \), where \([e]^{\text{TOP}} = \emptyset \vdash [f] \). But \( [[L \ \text{Size}(e)]]^{\text{TOP}} = \emptyset \vdash \square [f] \), which is the expected result. \( \Box \)

**Proposition 26 (Compositionality for multiple lift contexts)** If \( e \notin \text{Values} \), then

\[
[[F \ [e]]]^{\text{TOP}} = [[F \ \text{Size}(e) \ [e]]^{\text{TOP}}].
\]

183
Proof By induction on $F$. If $F = \Box$, there is nothing to prove. Otherwise, let $F = L [F']$ and $\zeta = \text{Size}(e)$.

By induction hypothesis, $[F'[e]]^\text{TOP} = [F']^\zeta_{\text{TOP}} [e]^\text{TOP}$.

As the Size function is compositional, $\zeta' = \text{Size}(F'[e]) = \text{Size}(F'[\Box])$.

By proposition 26, $[L [F'[e]]]^\text{TOP} = [L [F']^\zeta_{\text{TOP}} [F'[e]]]^\text{TOP} = [L [F']^\zeta_{\text{TOP}} [F'][e]^\text{TOP}]$.

By proposition 26, $[L [F'][\Box]^\zeta_{\text{TOP}} = [L [F'][\Box]^\zeta_{\text{TOP}} [F'][\Box]^\zeta_{\text{TOP}}] = [L [F'][\Box]^\zeta_{\text{TOP}} [F'][e]^\text{TOP}]$.

So, $[L [F'[e]]]^\text{TOP} = [L [F'][\Box]^\zeta_{\text{TOP}} [e]^\text{TOP}]$. □

**Lemma 40 (Compositionality for evaluation contexts)** If $e \not\in \text{Values}$, then

$$[E[e]]^\text{TOP} = [E]^\text{Size(e)} [e]^\text{TOP}.$$  

Proof By case on $E$. Let $\zeta = \text{Size}(e)$.

- If $E = F$, use proposition 26.

- If $E = b_v \vdash F$, then $[E[e]]^\text{TOP} = \text{TOP}(b_v) [[F[e]]^\text{TOP}]$

  $$= \text{TOP}(b_v) [[F]^\zeta_{\text{TOP}} [e]^\text{TOP}]$$

  $$= ([\text{TOP}(b_v) \circ [F]^\zeta_{\text{TOP}}] [e]^\text{TOP}]$$

  $$= [E]^\zeta_{\text{TOP}} [e]^\text{TOP}.$$  

- If $E = (b_v, x = F, b \vdash f)$, then let $b_c = ((x = F[e], b)$. We have $[E[e]]^\text{TOP} = \text{T Dum}(b_c) \circ \text{TOP}(b_v) \circ \text{TUp}(b_c) \{0 \vdash [f]\}$, since $F[e]$ cannot be a value.

Let $\Theta' \vdash E' = [[F[e]]]^\text{TOP} = [F]^\zeta_{\text{TOP}} [e]^\text{TOP}$ (by proposition 26).

Let $\Theta_u \vdash B = \text{TUp}(b)$, and $\zeta' = \text{Size}(F[e]) = \text{Size}(F[\Box])$.

Let $(x', \Psi') = \left\{ \begin{array}{ll} (x, E') & \text{if } \zeta' = \Box \\ (x, \text{update } x \ E') & \text{otherwise} \end{array} \right.$

We have $\text{TUp}(b_c) = \Theta_u + \Theta' \vdash x' = \Psi'[E'], B$. Let $\Psi_c = \Theta_u \vdash x' = \Psi' \vdash B \text{ in } [f]$. We have

$$[E[e]]^\text{TOP} = \text{T Dum}(b_c) \circ \text{TOP}(b_v) \circ \Psi_c \circ [F]^\zeta_{\text{TOP}} [e]^\text{TOP}$$

$$= [E]^\zeta_{\text{TOP}} [e]^\text{TOP}.$$  

□

When the expression filling the context hole is a value, we have seen that this compositionality property is false. We nevertheless prove a weaker one.

**Proposition 27 (Semi-compositionality for lift contexts)** For all evaluation context $\Psi$,

$$\Psi[[L]^\text{TOP} [v]^\text{TOP}]] \rightarrow^* \Psi[[L [v]^\text{TOP}]].$$  

Proof By case on $L$. Let $\zeta = \text{Size}(v)$ and $\Theta_v \vdash V = [v]^\text{TOP}$.

- If $L$ is of the shape $e' \Box$ or $\Box X$, then $\Psi[[L]^\text{TOP} [v]^\text{TOP}]] = \Psi[[L [v]^\text{TOP}]].$

- $L = \Box e$. Let $[e]^\text{TOP} = \Theta \vdash E$. We have $[e]^\zeta_{\text{TOP}} = \emptyset \vdash [e]^{\zeta_{\text{TOP}}}$ and $\Psi \circ [L]^\zeta_{\text{TOP}} [v]^\text{TOP}] = \Psi[\Theta_v \vdash V[e]]$, which by lemma 39 reduces to $\Psi[\Theta_v + \Theta \vdash VE] = \Psi[[L [v]^\text{TOP}]].$
• \( L = \{ s_v, X = \square, s \} \). Let \([s_v]^{\top} = \Theta_v + S_v', [s] = S \), and \([s]^{\top} = \Theta' + S' \).

We have \( \Psi \cdot [L]^{\top}([v])^{\top} = \Psi[\Theta_v + \Theta'_v + \{ S_v', X = V, S \}] \), which by lemma 39 reduces to \( \Psi[C] = \Psi[\Theta_v + \Theta'_v + \{ S_v', X = V, S' \}] \). If \( s \) is not evaluated, then \( C \) is exactly \([L[v]]^{\top} \).

Otherwise, \( \Psi[C] \) reduces by rule Context (Allocate) to \( \Psi[\Theta_v + \Theta'_v + \{ l \mapsto \{ S_v', X = V, S' \} \} + l] \), which is exactly \( \Psi[[L[v]]^{\top}] \).

\[ \square \]

**Proposition 28 (Semi-compositionality for multiple lift contexts)** For all evaluation context \( \Psi \),

\[ \Psi[[E]]^{\top}([v])^{\top} \rightarrow^{*} \Psi[[E[v]]^{\top}] \]

**Proof** By induction on \( E \). If \( E = \square \), there is nothing to prove. Otherwise, \( E = L[F'] \). Let \( \zeta = \text{Size}(v) \) and \( \zeta' = \text{Size}(F' [\square \zeta]) = \text{Size}(F' [v]) \) (by hypothesis 3).

By proposition 27, as neither \( F' [\square \zeta] \) nor \( F' [v] \) are values, we have \([E]^{\top} = [L]^{\top} \cdot [[F']^{\top}]^{\top} \) and \([E[v]]^{\top} = [L]^{\top} \cdot [[F'[v]]^{\top}]^{\top} \).

By induction hypothesis,

\[ \Psi \cdot [E]^{\top}([v])^{\top} = \Psi \cdot [L]^{\top} \cdot [[F']^{\top}]^{\top} \]

\[ \rightarrow^{*} \Psi \cdot [L]^{\top} \cdot [[F'[v]]^{\top}]^{\top} \]

\[ = \Psi[[E[v]]^{\top}] \]

\[ \square \]

**Proposition 29 (Semi-compositionality for evaluation contexts)** For all evaluation context \( \Psi \),

\[ \Psi[[E]]^{\top}([v])^{\top} \rightarrow^{*} \Psi[[E[v]]^{\top}] \]

**Proof** By case analysis on \( E \).

• \( E = (b_v \vdash F) \). Let \( \zeta = \text{Size}(v) \) and \( \zeta' = \text{Size}(F [\square \zeta]) = \text{Size}(F [v]) \) (by hypothesis 3). We have

\[ (\Psi \cdot [E]^{\top})([v])^{\top} \]

\[ = \Psi \cdot \text{Top}(b_v) \cdot [[F]^{\top}([v])]^{\top} \]

\[ \rightarrow^{*} \Psi \cdot \text{Top}(b_v) \cdot [[F[v]]^{\top}]^{\top} \]

(b by proposition 28)

\[ = \Psi[[b_v \vdash F[v]]^{\top}] \]

\[ = \Psi [[E[v]]^{\top}] \]

• \( E = (E F \vdash e) \), with \( B = (b_v, x = \square, b) \). Let \( \zeta = \text{Size}(v) \) and \( \zeta' = \text{Size}(F [\square \zeta]) = \text{Size}(F [v]) \) (by hypothesis 3). Let also \( b_0 = (x = \square, b) \). We have

\[ \Psi \cdot [E]^{\top}([v])^{\top} \]

\[ = \Psi \cdot \text{Dum}(b_0) \cdot \text{Top}(b_v) \cdot (TU_{p}(b_0) [\emptyset \vdash [e]]) \cdot [[F]^{\top}([v])^{\top}] \]

\[ \rightarrow^{*} \Psi \cdot \text{Dum}(b_0) \cdot \text{Top}(b_v) \cdot (TU_{p}(b_0) [\emptyset \vdash [e]]) \cdot [[F[v]]^{\top}]^{\top} \]

(by proposition 28)

If \( F[v] \) is not a value, the obtained expression is exactly \( \Psi[[E[v]]^{\top}] \). Otherwise, the obtained expression is a partial translation of \( E[v] \), so by proposition 23, it reduces to \( \Psi[[E[v]]^{\top}] \), as expected.

\[ \square \]

185
7.5.2 Translation of access

In \( \lambda_c \), the topmost binding is used as a heap, to store the values of variables. These values may then be copied when the corresponding bound variable is used in a strict context. In \( \lambda_{alloc} \), heaps can only contain blocks, i.e. records and functions. Variables (or constants if the calculus featured them) cannot be stored in them. Instead, we have seen that they are substituted on the fly during the translation. This distinction makes the translation of access a bit weird.

**Proposition 30** If \( TOP(b_v) = \Theta_a \vdash (\sigma, \eta) \), \( b_v(x) = v \), and \( [v]^{TOP} = \Theta_v \vdash V \), then \( \Theta_v \subseteq \Theta_a \) and \( (\sigma \circ \eta)(x) = V\{\sigma \circ \eta\} \).

**Proof** By induction on \( b_v \).

- \( b_v = \epsilon \). Contradicts \( b_v(x) = v \).
- \( b_v = (x = v, b_v') \) and \( \text{Size}(v) = n \). We have

\[
\begin{align*}
[v]^{TOP} &= \emptyset \vdash y = \Theta_v \vdash V \\
TOP(b_v') &= \Theta_a' \vdash (\sigma', \eta') \\
TOP(b_v) &= \Theta_v' + \Theta_a' \vdash (\sigma', \eta' + \{x \mapsto l\}) = \Theta_a \vdash (\sigma, \eta)
\end{align*}
\]

Obviously, we have \( \Theta_v \subseteq \Theta_a \). Furthermore, by syntactic correctness of \( b_v \), \( x \notin \text{dom}(\sigma) \), so \( (\sigma \circ \eta)(x) = \eta(x) = l = V = V\{\sigma \circ \eta\} \).

- \( b_v = (x = v, b_v') \), with \( \text{Size}(v') = \{?\} \). We have

\[
\begin{align*}
[v']^{TOP} &= \emptyset \vdash z = \Theta_v \vdash V \\
TOP(b_v') &= \Theta_a' \vdash (\sigma', \eta') \\
TOP(b_v) &= \Theta_v' + \Theta_a' \vdash (\sigma', \eta' + \{y \mapsto l\}) = \Theta_a \vdash (\sigma, \eta)
\end{align*}
\]

and therefore \( (\sigma \circ \eta)(x) = y\{\eta'\} = V\{\eta\} \).

- \( b_v = (y = v', b_v') \) and \( \text{Size}(v') = n \). We have

\[
\begin{align*}
[v']^{TOP} &= \emptyset \vdash l = \Theta_v \vdash V \\
TOP(b_v') &= \Theta_a' \vdash (\sigma', \eta') \\
TOP(b_v) &= \Theta_v' + \Theta_a' \vdash (\sigma', \eta' + \{y \mapsto l\}) = \Theta_a \vdash (\sigma, \eta)
\end{align*}
\]

By induction hypothesis, \( \Theta_v \subseteq \Theta_a' \), so \( \Theta_v \subseteq \Theta_a \). By induction hypothesis, \( (\sigma' \circ \eta')(x) = V\{\eta'\} \), so \( (\sigma \circ \eta)(x) = (\sigma' \circ \eta')(x)\{y \mapsto l\} = V\{\sigma' \circ \eta' \circ \{y \mapsto l\}\} = V\{\sigma \circ \eta\} \).

- \( b_v = (y = v', b_v') \) and \( \text{Size}(v') = \text{undefined} \). We have

\[
\begin{align*}
[v']^{TOP} &= \emptyset \vdash z = \Theta_v \vdash V \\
TOP(b_v') &= \Theta_a' \vdash (\sigma', \eta') \\
TOP(b_v) &= \Theta_v' + \Theta_a' \vdash (\sigma' \circ \{y \mapsto z\}, \eta') = \Theta_a \vdash (\sigma, \eta)
\end{align*}
\]

By induction hypothesis, \( \Theta_v \subseteq \Theta_a' \), so \( \Theta_v \subseteq \Theta_a \). By induction hypothesis, \( (\sigma' \circ \eta')(x) = V\{\eta'\} \). But by syntactic correctness of \( b_v \), we know that \( y \) is not free in \( b_v' \), so \( y \notin \text{cod}(\eta') \), and as we additionally have \( y \notin \text{dom}(\eta') \), we can deduce that \( \{y \mapsto z\} \circ \eta' = \eta' \circ \{y \mapsto z\} \{\eta'\} \).

So, we have

\[
\begin{align*}
(\sigma \circ \eta)(x) &= x\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\
&= x\{\sigma' \circ \eta' \circ \{y \mapsto z\} \{\eta'\}\} \\
&= (\sigma \circ \eta)(x)\{y \mapsto z\} \{\eta'\} \\
&= V\{\sigma' \circ \eta' \circ \{y \mapsto z\} \{\eta'\}\} \\
&= V\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\
&= V\{\sigma \circ \eta\}.
\end{align*}
\]

186
Proposition 31 (Access) Let $\Psi = \mathbb{E} \Gamma \top = \Theta \vdash \Phi[\sigma]$. If $\mathbb{E}(x) = v$ and $[v] \top = \Theta_v \vdash V$, then $\sigma(x) = V\{\sigma\}$ and $\Theta_v \subseteq \Theta$.

Proof By case analysis on the proof of $\mathbb{E}(x) = v$.

EA. $\mathbb{E} = b_v \vdash F$, and $b_v(x) = v$. We have

$$\mathbb{E} \Gamma \top = \top \vdash F \top = \top \vdash \Phi \top.$$

Let $\top \vdash \Phi \top = \Theta_v \vdash \Phi[\sigma]$ and $[\top] \top = \Theta \vdash \Phi[\sigma]$. We can deduce $\sigma = \sigma_v \circ \eta_a$. By proposition 30, we have $\Theta_v \subseteq \Theta_v \subseteq \Theta$ and $(\sigma \circ \eta_a)(x) = V\{\sigma_v \circ \eta_a\}$, or in other words $\sigma(x) = V\{\sigma\}$, which is the expected result.

IA. $\mathbb{E} = (b_v, x = F, b \vdash e)$. Then, $\mathbb{E} \Gamma \top = TDun(x = F[\top], b) \circ \top \vdash e \top$.

Let $\top \vdash e \top = \Theta \vdash e \top$ = $\top \vdash (\sigma_v \circ \eta_a)$

$\top \vdash e \top = \top \vdash (\sigma_v \circ \eta_a)$

$\top \vdash e \top = \top \vdash (\sigma_v \circ \eta_a)$

We have $\sigma = \sigma_v \circ \eta_a \circ \eta_d$. By proposition 30, $\Theta_v \subseteq \Theta_v$, so $\Theta_v \subseteq \Theta$. Furthermore, $(\sigma_v \circ \eta_a)(x) = V\{\sigma_v \circ \eta_a\}$, so $\sigma(x) = x\{\sigma_v \circ \eta_a \circ \eta_d\} = x\{\sigma_v \circ \eta_a\} \{\eta_d\} = V\{\sigma\}$, as expected.

\[\square\]

7.5.3 Translation of internal merging

Proposition 32 (Internal merging) If $b \vdash e \overset{\text{IM}}{\longrightarrow} b' \vdash e'$, then $[b \vdash e] \top \longrightarrow^* [b' \vdash e'] \top$.

Proof Let $b \vdash e = (b_v, x = (\text{let rec } b_1 \text{ in } e_1), b_1 \vdash f)$. Let $b_0 = (x = (\text{let rec } b_1 \text{ in } e_1), b_2)$ and $b'_0 = (x = e_1, b_2)$.

We have $[b \vdash e] \top = TDun(b_0) \circ \top \vdash e \top$.

Let now $(x', \Phi') = \{ (x, \square) \text{ if Size(e1) = Size(let rec b1 in e1) = [?]} (\text{cf hypothesis 3})

\{(y, \text{update x } \square) \text{ with y fresh otherwise.}\}

Let also $\Theta_1 \vdash E_1$ be defined as follows. If $b_1$ is evaluated, let $\Theta_1 \vdash E_1 = [e_1] \top$, and otherwise $\Theta_1 \vdash E_1 = \emptyset \vdash [e_1]$. This way, we always have $\text{let rec } b_1 \text{ in } e_1 \top = [b_1] \top \vdash \Theta_1 \vdash E_1$.

Finally, let $\Phi_1 = \emptyset \vdash x' = \Phi', \text{Update}(b_2)$ in $[f]$, and $b_1 = b_1, b'_1$, where $b'_1$ does not begin with a value. We have

$$TU_{\cdot}(b_1)[\emptyset \vdash f]$$

$$= \Phi_1 [b_1] \top \vdash \Theta_1 \vdash E_1$$

$$= \Phi_1 \circ TDun(b'_1) \circ \top \vdash e' \top.$$

But the context $TDun(b'_1) \circ \top \vdash e' \top$ is a weak evaluation context, and the domain of its substitution only concerns variables in the domain of $b_1$, which are disjoint from free variables in $b_2, f, x$ by the side condition to the rule IM. Therefore, this context commutes with $\Phi_1$, and

$$TU_{\cdot}(b_1)[\emptyset \vdash f]$$

$$= TDun(b'_1) \circ \top \vdash e' \top.$$

Now, if $b_1$ is not fully evaluated, the two translation are semantically identical. But if $b_1$ is fully evaluated, i.e. $b'_1 = e$, then $[b' \vdash e'] \top$ translates with the TOP translation until $e_1$, and possibly further, if $e_1$ is a value too. We distinguish the two cases.
1. $b_1$ is not fully evaluated. Let $TU_{b_1}^e = \Theta_{b_1} \vdash B_{b_1}^e$. We have $\Theta_1 \vdash E_1 = \emptyset \vdash [e_1]$ and with $\phi = TDum(b_0) \circ TOP(b_e) \circ TDum(b_1^e) \circ TOP(b_{e_1})$,

$$\begin{align*}
\llbracket b \vdash e \rrbracket^{\text{TOP}} &= \phi(\Theta_1 \vdash B_{b_1}^e \in [e_1][]) \cup \text{Update}(b_2) \in [f] \\
\text{LFP} \phi(\Theta_1 \vdash B_{b_1}^e \in [e_1][]) \cup \text{Update}(b_2) \in [f] \\
\text{EM} \phi(\Theta_1 \vdash [f'] \cup \text{Update}(b_2) \in [f] \\
= \phi(\Theta_1 \vdash \text{Update}(b_2) \vdash [f] \\
= \phi \circ TU_{b_1^e}(b_2, b_{e_1}^e) \llbracket 0 \vdash [f].
\end{align*}$$

But let us now examine $\phi$ a bit $TDum(b_0) \circ TOP(b_e) \circ TDum(b_1^e) \circ TOP(b_{e_1})$. First, notice that $TDum(b_0) = TDum(b_0^e)$, by hypothesis 3.

Then, $TOP(b_e)$ and $TDum(b_1^e)$ are two weak evaluation contexts, and the domain of the substitution of $TDum(b_1^e)$ is included in $dom(b_1^e)$, which is disjoint from the free variables of $b_0$, so if $TDum(b_1^e) = \Theta_d \vdash \eta_d^e$, then $TOP(b_e) \circ TDum(b_1^e) = \eta_d^e \circ TOP(b_e) \circ TDum(b_1^e)$. Moreover, $\eta_d^e$ is a variable allocation, and is therefore idempotent, so we can apply proposition 20 to obtain

$$\begin{align*}
\phi &= TDum(b_0) \circ TDum(b_1^e) \circ TOP(b_e) \circ TOP(b_{e_1}) \\
&= TDum(b_0^e) \circ TOP(b_e) \circ TOP(b_{e_1}) \\
&= TDum(b_0^e) \circ TOP(b_e) \circ TOP(b_{e_1}).
\end{align*}$$

Furthermore, $TOP(b_{e_1}) = \Theta_{b_{e_1}} \vdash (\sigma_{b_{e_1}}, \eta_{b_{e_1}})$, where $\eta_{b_{e_1}}$ is idempotent, we have $TOP(b_{e_1}) = \eta_{b_{e_1}} \circ TOP(b_{e_1})$. But we know that the domain of $\eta_{b_{e_1}}$ is disjoint from the free variables of $TOP(b_e)$, so $TOP(b_e) \circ \eta_{b_{e_1}} = \eta_{b_{e_1}} \circ TOP(b_e)$, and therefore $\phi = TDum(b_1^e, b_{e_1}^e) \circ \eta_{b_{e_1}} \circ TOP(b_{e_1})$. But by corollary 10, $\eta_{b_{e_1}} \circ TOP(b_e) \circ TOP(b_{e_1}) = TOP(b_e, b_{e_1})$, so $\phi = TDum(b_1^e, b_{e_1}^e) \circ TOP(b_e, b_{e_1})$.

Finally, we obtain that

$$\begin{align*}
\llbracket b \vdash e \rrbracket^{\text{TOP}} &= TDum(b_0) \circ TOP(b_e) \circ TOP(b_{e_1}) \circ \Phi_1[\Theta_1 \vdash E_1].
\end{align*}$$

2. $b_1$ is fully evaluated. We have $\llbracket b \vdash e \rrbracket^{\text{TOP}} = TDum(b_0) \circ TOP(b_e) \circ TOP(b_{e_1}) \circ \Phi_1[\Theta_1 \vdash E_1]$.

Let $TOP(b_{e_1}) = \Theta_{b_{e_1}} \vdash (\sigma_{b_{e_1}}, \eta_{b_{e_1}})$. We know that $\eta_{b_{e_1}}$ is idempotent, so $TOP(b_{e_1}) = \eta_{b_{e_1}} \circ TOP(b_{e_1})$. As above, $dom(\eta_{b_{e_1}}) \perp \text{FV}(TOP(b_e))$, so $TOP(b_e) \circ TOP(b_{e_1}) = \eta_{b_{e_1}} \circ TOP(b_e) \circ TOP(b_{e_1})$, in which by corollary 10 we recognize $TOP(b_e, b_{e_1})$.

Therefore, $\llbracket b \vdash e \rrbracket^{\text{TOP}} = TDum(b_0) \circ TOP(b_e) \circ TOP(b_{e_1}) \circ \Phi_1[\Theta_1 \vdash E_1].$

But we notice that $\Phi_1[\Theta_1 \vdash E_1] = TU_{b_1}([b_0] \llbracket 0 \vdash [f] \llbracket])$. And by hypothesis 3, $TDum(b_0) = TDum(b_0^e)$. Let $TDum(b_0) = \Theta_{b_0} \vdash \eta_{b_0}$. By proposition 20, we have $TDum(b_0) \circ TOP(b_e, b_{e_1}) = \eta_{b_0} \circ TOP(b_e, b_{e_1}) \circ TDum(b_0^e) \circ TOP(b_{e_1}) \circ Update(b_2) \llbracket 0 \vdash [f] \llbracket]$. Let $b_2^e = (b_2, b_2^e)$, with $b_2^e$ not beginning with a value. By proposition 21, $\llbracket b \vdash e \rrbracket^{\text{TOP}} \rightarrow^* \eta_{b_0} \circ TOP(b_e, b_{e_1}) \circ TDum(b_0^e) \circ Update(b_2^e) \llbracket 0 \vdash [f] \llbracket]$.

But if $TDum(b_0) = \Theta_{b_0} \vdash \eta_{b_0}$ and $TDum(b_0^e) = \Theta_{b_0}^e \vdash \eta_{b_0}^e$, then $\eta_{b_0} = \eta_{b_0} + \eta_{b_0}^e$, so by proposition 20, the obtained expression is equal to $\eta_{b_0} \circ Update(b_2^e) \llbracket 0 \vdash [f] \llbracket].$ But $\eta_{b_0}$ commutes with $TDum(b_0^e)$, so we obtain $TDum(b_0^e) \circ \eta_{b_0} \circ TOP(b_e, b_{e_1}) \circ Update(b_2^e) \llbracket 0 \vdash [f] \llbracket]$, which by corollary 10 is equal to $TDum(b_0^e) \circ \eta_{b_0} \circ TOP(b_e, b_{e_1}) \circ Update(b_2^e) \llbracket 0 \vdash [f] \llbracket]$, which is exactly $\llbracket b \vdash e' \rrbracket^{\text{TOP}}$.
• Evaluated binding contexts
\[ \mathbb{B}_e := b_{v_1}, x = \Box, b_{v_2} \quad \text{with} \quad \text{Depth}(b_{v_1}, x = \Box, b_{v_2}) \text{ defined as } 1 + | b_{v_1} | \]

• Depth of an evaluation context

\[
\begin{align*}
\text{Depth}(\Box) &= 0 \\
\text{Depth}(\mathbb{L}[\mathbb{F}]) &= 1 + \text{Depth}(\mathbb{F}) \\
\text{Depth}(b_v \vdash \mathbb{F}) &= 1 + | b_v | + \text{Depth}(\mathbb{F}) \\
\text{Depth}(\mathbb{B}_e[\mathbb{F}] \vdash e) &= \text{Depth}(\mathbb{B}_e) + \text{Depth}(\mathbb{F})
\end{align*}
\]

• Measuring the number of let rec nodes
\( \mu_l(e) \) is the number of let rec nodes not under a \( \lambda \) in \( e \) (same for configurations).

• Measuring the depth of the let rec to lift (same for configurations)
\[
\begin{align*}
\mu_d(\mathbb{L}[\text{let rec } b_v \text{ in } e]) &= 1 + \text{Depth}(\mathbb{F}) \\
\mu_d(e) &= 0 \text{ otherwise}
\end{align*}
\]

well defined since the sum of the depths of let rec nodes strictly decreases.

• Measuring the binding level of the hot variable
\( \mu_b(e) \) is the depth of the binder of the hot variable, if any:
\[
\begin{align*}
\mu_b(\mathbb{B}_e[v], y = \mathbb{F}[x], b \vdash e) &= \text{Depth}(\mathbb{B}_e) \text{ if } (\mathbb{B}_e[v])(x) = v \\
\mu_b(\mathbb{B}_e[v] \vdash \mathbb{F}[x]) &= \text{Depth}(\mathbb{B}_e) \text{ if } (\mathbb{B}_e[v])(x) = v \\
\mu_b(e) &= 0 \text{ otherwise}
\end{align*}
\]

• Measure
\[
\begin{align*}
\mu_{\mu}(e) &= (\mu_l(e), \mu_d(e)) \quad \text{(lexicographically ordered).} \\
\mu(c) &= (\mu_l(c), \mu_d(c), \mu_b(c))
\end{align*}
\]

Figure 7.21: Measure

7.5.4 Simulation

Due to their different ways of handling bindings, the two calculi \( \lambda_c \) and \( \lambda_{alloc} \) do not yield a step by step simulation. Indeed, a redex and its reduct in \( \lambda_c \) may have the same translation. As an example, consider any expressions of the shape \( \mathbb{L}[\text{let rec } b_v \text{ in } e] \) and \( \text{let rec } b_v \text{ in } \mathbb{L}[e] \). The binding \( b_v \) is translated as a heap \( \Theta \) and a substitution \( \sigma_r \), in both cases, and the fact that it is under or above the \( \mathbb{L} \) context is not visible in the translation. The only problem with this is that in some cases an infinite reduction sequence in \( \lambda_c \) could be translated as an empty one in \( \lambda_{alloc} \), thus possibly changing the infinite loop observable behavior. In order to ensure that this doesn’t happen, we prove that such silent reduction steps cannot happen indefinitely. For this, we introduce a measure on expressions and configurations that strictly decreases during silent reductions steps. Its definition is given in figure 7.21.

It first defines two functions from expressions to \( \mathbb{N} \). The first, \( \mu_l \), is the number of let rec nodes not under a lambda in the given expression. The second, \( \mu_d \), is the depth of the let rec node to lift in the given expression, if any. Formally, if \( e \) is of the shape \( \mathbb{L}[\text{let rec } b \text{ in } f] \), then the let rec node can be lifted by rule \text{Lift}, so the result is the depth of the context \( \mathbb{L}[L] \), or 1 plus the depth of \( F \).

The functions \( \mu_l \) and \( \mu_d \) form a measure \( \mu_{\mu} \) on expressions, defined by \( \mu_{\mu} = (\mu_l, \mu_d) \), ordered lexicographically.

Moreover, these two functions are straightforwardly extended to configurations, replacing \( \mathbb{F} \) with \( E \) for the second definition.

A third function \( \mu_b \) is defined, but only on configurations, giving the depth of the binder for the hot variable, if any. We say that \( x \) is the hot variable in \( c \) if \( c \) is of the shape \( \mathbb{E}[\mathbb{N}[x]] \). Then \( \mu_b(e) \) is the depth at which \( x \) is bound in \( E \). Formally, we define evaluated binding contexts as

189
binding contexts of the shape \( b_{v_1}, x = \Box, b_{v_2}, \) and their depth as 1 plus the cardinal of \( b_{v_1}. \) Then the depth of multiple lift contexts is defined as the number of nested lift contexts, and the depth of evaluation contexts is defined accordingly.

A property of this measure is that it is monotone through contextual closure.

**Proposition 33** If \( \mu_e(c) > \mu_e(c'), \) then for any evaluation context \( \mathcal{E}, \mu(\mathcal{E}[c]) > \mu(\mathcal{E}[c']). \)

**Proof** The property clearly holds for both measures \( \mu_\lambda \) and \( \mu_\Delta, \) thus for their lexicographic product as well. \( \Box \)

**Lemma 41 (Contraction simulated)** If \( e \leadsto e', \) then \( \llbracket e \rrbracket_{\text{TOP}} \rightarrow \llbracket e' \rrbracket_{\text{TOP}} \) or \( \llbracket e \rrbracket_{\text{TOP}} = \llbracket e' \rrbracket_{\text{TOP}} \) and for any \( \mathcal{E}, \mu(\mathcal{E}[c]) > \mu(\mathcal{E}[c']). \)

**Proof** By case analysis on the applied rule.

**Beta.** \( e = ((\lambda x)[f]), \) and \( e' = \text{letrec} \ x = v \ f. \) Let \( \llbracket e \rrbracket_{\text{TOP}} = \Theta_e \vdash V. \) We have \( \llbracket e \rrbracket_{\text{TOP}} = \Theta_e + [l \mapsto (\lambda x. [f])] \vdash l \ V, \) which reduces by rule **Beta** to \( \Theta_e + [l \mapsto (\lambda x. [f])] \vdash f \ x \rightarrow V. \)

Let us now calculate \( \text{TOP}(x = v). \)

- If \( \text{Size}(v) = [?], \) then \( \Theta_e \vdash V = \emptyset \vdash V, \) and \( \text{TOP}(x = v) = \emptyset \vdash (x \rightarrow V, \text{id}); \) so \( \llbracket x = v \rrbracket_{\text{TOP}} = \emptyset \vdash \Box [x \rightarrow V] = \Theta_e \vdash \Box [x \rightarrow V]. \)
- Otherwise, \( \Theta_e \vdash V = \Theta_e \vdash l, \) and \( \text{TOP}(x = v) = \Theta_e \vdash (\text{id}, x \rightarrow l); \) so \( \llbracket x = v \rrbracket_{\text{TOP}} = \Theta_e \vdash \Box [x \rightarrow l] = \Theta_e \vdash \Box [x \rightarrow V]. \)

So, in both cases, we have \( \llbracket x = v \rrbracket_{\text{TOP}} = \Theta_e \vdash \Box [x \rightarrow V]. \) Therefore, \( \llbracket x = v \rrbracket_{\text{TOP}} \) reduces to \( \llbracket x = v \rrbracket_{\text{TOP}} [f], \) which by lemma 39 reduces to \( \llbracket x = v \rrbracket_{\text{TOP}} [f_{\text{TOP}}], \) which is exactly \( \llbracket e' \rrbracket_{\text{TOP}}. \)

**Project.** \( e = \{s_e\}. X \) and \( e' = s_e(X). \) Let \( s_e = (X_1 = v_1 \ldots X_n = v_n), \) \( X = X_{i_0}, \) and for each \( i, \llbracket s_e \rrbracket_{\text{TOP}} = \Theta_i \vdash V_i. \) We have \( \llbracket s_e \rrbracket_{\text{TOP}} = \bigcup_{1 \leq i \leq n} \Theta_i + (X_1 = V_1 \ldots X_n = V_n), \) and

\[
\llbracket \llbracket e \rrbracket_{\text{TOP}} = \bigcup_{1 \leq i \leq n} \Theta_i + \{l \mapsto \{X_1 = V_1 \ldots X_n = V_n\} \} \vdash l. X. \]

By rule **Project**, it reduces to

\[
\bigcup_{1 \leq i \leq n} \Theta_i + \{l \mapsto \{X_1 = V_1 \ldots X_n = V_n\} \} \vdash V_{i_0}, \]

which by rule **GC** reduces to \( \Theta_{i_0} \vdash V_{i_0}, \) which is exactly \( \llbracket e' \rrbracket_{\text{TOP}}. \)

**Lift.** \( e = L \llbracket \text{let rec } b \text{ in } f \rrbracket \) and \( e' = \text{let rec } b \text{ in } L[f]. \)

- If \( b \) is evaluated, then \( \llbracket e \rrbracket_{\text{TOP}} = [L][f] \llbracket \text{TOP(b)} \rrbracket [f_{\text{TOP}}]. \) Let \( \text{TOP}(b) = \Theta \vdash (\Box, \sigma). \) In the context \( [L][f] \llbracket \text{TOP(b)} \rrbracket, \) the only substitution is \( \sigma, \) whose domain is \( \text{dom}(b), \) which by the side condition to the **Lift** rule is disjoint from the free variables of \( L, \) so the contexts commute, and \( \llbracket e \rrbracket_{\text{TOP}} = \text{TOP}(b) \llbracket [L][f]_{\text{TOP}} \rrbracket = \llbracket e' \rrbracket_{\text{TOP}}. \)
- If \( b \) is not evaluated, then \( b = b_e, b', \) with \( b' \) non empty and not beginning with a value. We have \( \llbracket e \rrbracket_{\text{TOP}} = [L][f] \llbracket \text{TDum}(b') \rrbracket \llbracket \text{TOP}(b_e) \rrbracket \llbracket \text{TP}(b') \rrbracket [\emptyset \vdash [f]]. \) But as above, the context \( [L][f] \llbracket \text{TOP} \rrbracket \) has not substitution and is not affected by the ones of \( \text{TDum}(b') \), \( \text{TOP}(b_e), \) and \( \text{TP}(b'). \) So \( \llbracket e \rrbracket_{\text{TOP}} = \text{TDum}(b') \llbracket \text{TOP}(b_e) \rrbracket \llbracket \text{TP}(b') \rrbracket [L][f]_{\text{TOP}} [\emptyset \vdash [f]] = \llbracket e' \rrbracket_{\text{TOP}}. \)

This is the only case where the two translations are directly equal. We thus have to show that \( \mu_\Delta(e) > \mu_\Delta(e'). \) And indeed \( \mu_\Delta(e) = \mu_\Delta(L \llbracket \text{let rec } b \text{ in } f \rrbracket) = 2 + 0, \) whereas \( \mu_\Delta(e') = \mu_\Delta(\text{let rec } b \text{ in } L[f]) = 0. \) Conclude by proposition 33.
There is a last difficulty lying in the way to the theorem of simulation, due to different sharing properties of the two calculi. Consider the configuration \( c = (x = \lambda y.y) \vdash (x.x)x \). It reduces by rule \( \text{SUBST} \) to \( c' = (x = \{X = \lambda y.y\} \vdash \{X = \lambda y.y\}.x)x \). By the \text{TOP} translation, \( c \) is translated to a configuration

\[
C = \left\{ \begin{array}{l}
  l_1 \mapsto \lambda y.y, \\
  l_2 \mapsto \{X = l_1\}
\end{array} \right\} \vdash (l_2.X)l_2.
\]

By the same translation, \( c' \) is translated to a configuration

\[
C' = \left\{ \begin{array}{l}
  l_1 \mapsto \lambda y.y, \\
  l_2 \mapsto \{X = l_1\}, \\
  l_3 \mapsto \lambda y.y, \\
  l_4 \mapsto \{X = l_3\}
\end{array} \right\} \vdash (l_4.X)l_2.
\]

The heap \( \Theta' \) of \( C' \) contains an additional copy of the record and the function. This phenomenon happens at each application of the \( \text{SUBST} \) rule. But except in case of a faulty configuration (see below), such a reduction step is necessarily followed by a \( \text{BETA} \) or a \( \text{PROJECT} \) step. In our example, a \( \text{PROJECT} \) step occurs, that destroys the copied record: \( c' \) reduces to \( c'' = (x = \{X = \lambda y.y\} \vdash (\lambda y.y)x) \). This reduction step destroys the copied record immediately after it has been copied. Similarly, when a function is copied, it is immediately destroyed by a \( \text{BETA} \) reduction step. In both cases, the translated configuration reduces in one step, by the same rule (\( \text{PROJECT} \) or \( \text{BETA} \)). As a consequence, our simulation theorem takes this possibility into account, and allows a couple of successive reductions steps to be simulated by a single one.

But this is not yet sufficient. Indeed, in the case of the \( \text{PROJECT} \) rule, not only the record is duplicated, but also the values it contains. In our example, the function \( \lambda y.y \) is copied. And even after applying the \( \text{PROJECT} \) rule, it remains, as shown by the translation of \( c'' \):

\[
C'' = \left\{ \begin{array}{l}
  l_1 \mapsto \lambda y.y, \\
  l_2 \mapsto \{X = l_1\}, \\
  l_3 \mapsto \lambda y.y
\end{array} \right\} \vdash l_3l_2.
\]

Our solution to this problem consists in only considering expressions where all the record fields are variables, which we call \( \text{R} \)-normal expressions. Any expression can be transformed into an \( \text{R} \)-normal one, by applying the following \( \text{NAMEFIELDS} \) rule, in any context.


\[
\frac{\exists i, e_i \notin \text{Vars}}{\{X_1 = e_1 \ldots X_n = e_n\} \text{ let rec } x_1 = e_1 \ldots x_n = e_n \text{ in } \{X_1 = x_1 \ldots X_n = x_n\}} \quad \text{[NAMEFIELDS]}
\]

This process necessarily terminates since the number of records not containing only variables strictly decreases. The reduction rules of \( \lambda_c \) obviously preserve the \( \text{R} \)-normality. This way, after a sequence of a \( \text{SUBST} \) step followed by a \( \text{PROJECT} \) step, no duplication has been made: an expression of the shape \( x.X \) has been replaced with another variable.

We can now state our final theorem. A \( \lambda_c \) configuration is said \textit{stuck on a free variable} when it is of the shape \( E \in \text{N}[x] \) and \( E(x) \) is undefined. This definition is extended to \( \lambda_{\text{alloc}} \) configurations (replace \( E \) with \( \Psi \)). We say that a configuration is faulty if it is in normal form and is not a valid answer and is not stuck on a free variable. Roughly, the theorem states that if a configuration \( c \) reduces to another one \( c' \), then

- either \( c' \) is faulty and so is the translation of \( c \),
- or the translation of \( c \) reduces to the one of \( c' \),

\[191\]
• or $c'$ itself reduces to $c''$, such that the translation of $c$ reduces to the one of $c''$.
• or $c$ and $c'$ are translated to the same configuration, but $\mu(c) > \mu(c')$.

This complicated result is due to the fact that $\lambda_c$ first needs to duplicate a function before to apply it, and to duplicate a record before to select a component from it, and to the fact that the TOP translation identifies some configurations, by performing some lifting and merging steps by itself.

**Theorem 5 (Small steps encoding)** For all $R$-normal configuration $c$, if $c \rightarrow c'$ and $[c]^{\text{TOP}} = C$, then one of the four situations below holds:

1. Either $c'$ is faulty, and then $C$ is faulty too;

   $$
   \begin{array}{c}
   \text{c} \\
   \downarrow \\
   \text{C}
   \end{array}
   \rightarrow
   \begin{array}{c}
   \text{c'} \\
   \downarrow \\
   \text{C'}
   \end{array}
   $$

2. or there exists $C'$ such that $[c'] = C'$ and $C \rightarrow^+ C'$;

   $$
   \begin{array}{c}
   \text{c} \\
   \downarrow \\
   \text{C}
   \end{array}
   \rightarrow
   \begin{array}{c}
   \text{c'} \\
   \downarrow \\
   \text{C'}
   \end{array}
   $$

3. or there exists $c''$, $C'$ such that $[c''] = C'$ and $C \rightarrow^+ C'$;

   $$
   \begin{array}{c}
   \text{c} \\
   \downarrow \\
   \text{C}
   \end{array}
   \rightarrow
   \begin{array}{c}
   \text{c'} \\
   \downarrow \\
   \text{C'}
   \end{array}
   $$

4. or $[c'] = C$ directly, and $\mu(c) > \mu(c')$

   $$
   \begin{array}{c}
   \text{c} \\
   \downarrow \\
   \text{C}
   \end{array}
   \rightarrow
   \begin{array}{c}
   \text{c'} \\
   \downarrow \\
   \text{C'}
   \end{array}
   $$

**Proof** By case analysis on the applied rule.

**Context.** By lemma 41.

**IM.** By proposition 32, noting that the number of let rec nodes decreases by one when applying the rule.

**EM.** $c = b_c \vdash \text{let rec } b \text{ in } c \text{ and } c' = b_{c'}, b \vdash e$. Let us now define $C_1$ by $\emptyset \vdash [e]$ if $b$ is not evaluated, and $[c]^{\text{TOP}}$ otherwise. Then $[c]^{\text{TOP}} = \text{TOP}(b_c) \circ [b]^{\text{TOP}}[C_1]$. Let $b = b_{c'}, b'$, where $b'$ does not begin with a value. We have $[c]^{\text{TOP}} = \text{TOP}(b_c) \circ \text{TDum}(b') \circ \text{TOP}(b_{c'}) \circ \text{TUp}(b'[C_1])$. But the substitution of the context $\text{TDum}(b')$ does not affect $\text{TOP}(b_c)$ and conversely the substitution of $\text{TOP}(b_c)$ does not affect $\text{TDum}(b')$, so the two contexts commute. But then $\text{TOP}(b_{c'})$ is next to $\text{TOP}(b_c)$. Let $\eta$ be the substitution of $\text{TDum}(b_{c'})$. It does not affect $\text{TOP}(b_{c'})$, by the side condition to the EM rule, so $\text{TOP}(b_c) \circ \text{TOP}(b_{c'}) = \eta \circ \text{TOP}(b_{c'}) \circ TUp(b'[C_1]) = \eta \circ \text{TOP}(b_{c'}) | C_1 = \text{TOP}(b_{c'}) | C_1$. The number of let rec nodes again decreases by one.

**Subst.** $c = E [N [x]], c' = E [N [v]]$, and $E (x) = v$. Let $\Psi = [E]^{\text{TOP}} = \emptyset \vdash \Phi [\pi]$.

- If $v$ is a variable $y$, then $[v]^{\text{TOP}} = \emptyset \vdash y$, and by proposition 31, $\sigma(x) = y[\sigma]$, so $[c]^{\text{TOP}} = [c']^{\text{TOP}}$. But, the depth of the binder of the hot variable, from the depth of $x = y$ in $E$, becomes either an upper $y = c'$ definition, or the depth 0, if $y$ is not defined by $E$, so $\mu(c) > \mu(c')$.
- If $c'$ is faulty, i.e. either $N = \Box c'$ and $v$ is a record, or $N = \Box X$ and $v$ is a function or a record with no $X$ field, then $C$ is faulty too.
• If \( v = \lambda_y e \) and \( \mathbb{N} = \square \nu' \), then \( c' \xrightarrow{} c'' = \mathbb{E} \{ \text{let } y = \nu' \text{ in } e \} \).

Let \([e']^\top = \Theta_v \vdash V'\). Let \( \phi = \Theta'_v \vdash \square[\text{id}] \). We have \( C = \Psi \circ \phi[y \rightarrow V'] \).

But by proposition 31, the location \( l = \sigma(x) \) is such that \( \Theta(l) = \lambda y \cdot [e] \). Therefore, \( C \) reduces by rule CONTEXT (BETA) to \( \Psi \circ \phi[e]^{\top} \{ y \rightarrow V' \} \). By lemma 39, this reduces to \( \Psi \circ \phi[e]^{\top} \{ y \rightarrow V' \} \).

Let now \( \phi' = \phi \circ \{ y \rightarrow V' \} \). The obtained configuration can be written \( \Psi \circ \phi'[e]^{\top} \).

But \( \text{TOP}(y = \nu') = \Theta'_v \vdash \square[\{ y \rightarrow V' \}] = \phi', \) so \( \{ \text{let } y = \nu' \text{ in } e \}^{\top} = \phi'[e]^{\top} \), and the obtained term can also be written \( [\mathbb{E}]_{\text{TOP}} \{ \text{let } y = \nu' \text{ in } e \}^{\top} \), which by proposition 29, reduces to \( [\mathbb{E}]_{\text{TOP}} \{ \text{let } y = \nu' \text{ in } e \}^{\top} \), which is exactly \( [c']^{\top} \).

• If \( v = \{ s_v \}, \mathbb{N} = \square X \), with \( X \in \text{dom}(s_v) \), then \( c' \xrightarrow{} c'' = \mathbb{E} [s_v(X)] \).

By hypothesis, \( c \) is in \( \mathbb{R} \)-normal form, so there exist names \( X_1 \ldots X_n \) and variables \( x_1 \ldots x_n \) such that \( s_v = (X_1 = x_1 \ldots X_n = x_n) \). Then, \( s_v \) can be viewed as a record of \( \lambda_{\text{alloc}} \), and \( [v]^{\top} = \{ l \mapsto \{ s_v \} \} \vdash l \).

By proposition 31, we have \( \sigma(x) = l \) and \( \Theta(l) = \{ s_v \} \). We have \( [e]^{\top} = \Psi[x, X] = \Psi[l, X] \). As \( e \) reduces to \( c' \), there exists an index \( i_0 \) such that \( X = X_{i_0} \). So, \( [e]^{\top} \) reduces in one PROJECT step to \( \Psi[X_{i_0}] \), which is \( [\mathbb{E}]_{\text{TOP}} \{ [X_{i_0}]^{\top} \} \), so by lemma 39, it reduces to \( [\mathbb{E}]_{\text{TOP}} \{ [X_{i_0}] \} \), which is exactly the translation of \( c'' \).

\( \square \)

Eventually, we state a less precise theorem, more like what we would obtain with big step semantics.

**Theorem 6 (Big steps encoding)**

1. For all expression \( e \), if \( \emptyset \vdash e \longrightarrow^* a \), then \( \emptyset \vdash [e] \longrightarrow^* [a] \).
2. For all expression \( e \), if \( e \) goes wrong, i.e. \( \emptyset \vdash e \) reduces to a faulty configuration, then \( [e] \) also goes wrong.
3. For all expression \( e \), if \( e \) loops, i.e. there exists an infinite reduction sequence starting from \( \emptyset \vdash e \), then \( [e] \) also loops.
4. For all expression \( e \), if \( e \) gets stuck on a free variable, then so does \( [e] \).

**Proof** For items 1 and 2, notice that \( \emptyset \vdash [e] \) reduces to \( [e]^{\top} \), and then reason by induction on the length of the reduction sequence. For item 3, by contrapositive: we know that there is a reduction sequence in \( \lambda_{\text{alloc}} \) simulating the one in \( \lambda_e \), but it could be of phantom steps, i.e. the same configuration could be a translation for all steps. However this would contradict the strict decreasing of the measure, which is of course bounded by 0. For item 4, the reduction leading to the configuration stuck on a free variable is simulated, and the reached configuration being the translation of a stuck configuration is also stuck. \( \square \)

The initial goal here was to prove the correctness of our compilation scheme, but in fact we have a completeness result for free.

**Theorem 7 (Big steps completeness)**

1. If \( \emptyset \vdash [e] \longrightarrow^* A \), then there exists a such that \( \emptyset \vdash e \longrightarrow^* a \) and \( [a]^{\top} = A \).
2. If \( [e] \) goes wrong, then \( e \) also goes wrong.
3. If \( [e] \) loops, then \( e \) also loops.
4. If \( [e] \) gets stuck on a free variable, then so does \( e \).
Proof There are four possible final states for a configuration: it can reduce to a value, or it can get stuck on a free variable, or it can go wrong, or it can loop. We know that if a configuration \( \emptyset \vdash e \) reaches a final state, then so does \( [\emptyset \vdash e]^\text{top} \). But the four possible final states are mutually exclusive. Therefore, if the translation of an expression reaches a final state, then the original configuration necessarily reaches the same one. □

Remark 3 (Free variables) Free variables do not appear during reduction, and the cases where the evaluation gets stuck on a free variable do not occur if the initial expression is closed.

7.6 Related work

Cyclic explicit substitutions In [64], Rose defines a calculus with mutually recursive definitions, where the dedicated construct for recursion is presented as explicit cyclic substitution, referring to the explicit substitutions of Lévy et al. [2]. Instead of lifting recursive bindings to the top of terms as we do, the calculus pushes them inside terms, as usual with explicit substitutions. This results in the loss of sharing information. Any term is allowed in recursive bindings, but inside a recursive binding, when computing a definition, it is not possible to use the value of any definition from the same binding. In \( \lambda_c \), the rule for substitution \( \text{Subst} \) allows this, in conjunction with the internal access rule \( \text{IA} \). In Rose’s calculus, correct call by value reduction requires that in any binding, recursive definitions reduce to values, without really using each other. In this respect, it is less powerful than \( \lambda_c \). Besides, it does not impose size constraints on definitions, but is also not concerned with data representation.

Lescanne et al. [9] study sharing and different evaluation strategies, with a slightly different notion of cyclic explicit substitution. Any term is accepted in a recursive definition, but instead of going wrong when the recursive value is really needed, as in our system, the system of [9] loops. The focus of the paper is on the comparison between \( \lambda \)-graph reduction and environment based evaluation, and different evaluation strategies. No emphasis is put on data representation either.

Equational theories of the \( \lambda \)-calculus with explicit recursion Ariola et al. [7] study a \( \lambda \)-calculus with explicit recursion. Its semantics is given by source-to-source rewrite rules, where let rec is lifted to the top of terms, and definitions in a binding may use each other, as in \( \lambda_c \). The semantics of our source language \( \lambda_c \) is largely inspired by their call-by-value calculus, as a quite straightforward specialization of it. Thus, our work can be seen as importing the internal substitution rule \( \text{IA} \) from equational theory to language design. Nevertheless, the concerns are different: we deal with implementation and data representation, while Ariola et al. rather examine confluence, sharing and different evaluation strategies, including strong reduction (reduction under \( \lambda \)-abstraction).

let rec for objects and mixin modules Boudol’s construct [12], or Hirschowitz and Leroy’s [45], are different from the one of \( \lambda_c \) in several aspects. First, they accept strictly more expressions as recursive definitions. For instance, Boudol’s semantics of objects makes an extensive use of recursive definitions such as \( \text{let rec } o = \text{generator}(o) \) in \( e \). Such definitions are impossible in \( \lambda_c \). However, \( \lambda_c \) allows to define in the same binding some recursive values, followed by computations using these values. The semantics of mixin modules [47] requires complex sequences of alternate recursive and non-recursive bindings, which are trivial to write in \( \lambda_c \). On the whole, the loss of flexibility for valid recursive definitions allows to improve efficiency, thanks to the loss of additional indirections.

We believe that it is possible to combine the ideas of [12] and [47]. Consider a language where a recursive definition can be of any shape, and can now be syntactically annotated with integers representing its expected size. This language can be compiled exactly as \( \lambda_c \), but it features a more
powerful let rec construct. The idea should be seen as a compilation technique for Boudol's objects and Hirschowitz and Leroy's mixin modules, where the necessary size informations are statically available.
Chapter 8

Untyped compilation with local definitions
Future work and conclusions
Index

C
- calculus of constructions, 101
- carrier, 113
- collections, 113
- concrete dynamic graph, 90
- constructed values, 101
- constructors, 99, 101

D
data type definition, 101

E
- entities, 113
- equi-recursive, 100

I
- inductive types, 101
- interfaces, 113
- iso-recursive, 100

M
- methods, 113
- module-recursive references, 112

N
- nodes, 90, 93

O
- opaque, 99
- opaque fixed-point, 111

P
- partial matchings, 101
- phase-splitting, 110

R
- recursively dependent signature, 111
- representation independence, 99
- representation type, 113

S
- Shao’s equation, 100
- species, 113
- static dependency graph, 90

T
- translucent sums, 98
- transparent, 99
- transparent fixed-point, 111

Type paths, 101
- type unfolding, 101

V
- virtually concrete, 46
Bibliography


