Call-by-value mixin modules

Tom Hirschowitz

March 21, 2003

Contents

A faire

Graph subtyping In the rule 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 ν quantifiers? Types should be computed and ν really has the effect of creating a new variable that can be extruded to the top level. A question is where exactly should the ν computations be suspended, i.e. what are the lambdas for ν, under which nothing happens? Generative functors for sure, at least. Unfortunately, Sewell's interpreteation probably does not account for applicative ones anyway.

On Duggan and Sourelis The method of Duggan and Sourelis 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.

Ohter problems in Sourelis' 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 (implicitely 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:

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

```
end

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

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

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 $\{\mathit{CMS}, \mathbf{m}, \mathit{CMS}_v, \mathit{MM}, \mathit{MML}, \ldots\}$, 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 $\langle l_1 \triangleright x_1 \ldots l_n \triangleright x_n; \mathit{RES} \triangleright e\rangle$. Labels are mandatory in function applications $(e\ l_{i_1}{:}e_{i_1} \ldots l_{i_n}{:}e_{i_n})$, which are encoded as

$$(\mathsf{close}(e + \langle \epsilon; l_{i_1} \triangleright e_{i_1} \ldots l_{i_n} \triangleright e_{i_n}\rangle)).\mathit{RES}$$

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 τ and τ', respectively, and assume g needs f to be of type τ at most. Covariant subtyping of methods can leed to giving f a type τ'', super type of τ. But then, f can be overrriden by a method of type τ'', 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 τ, insert a coercion to type τ. Graph subtyping and inputs subtyping can be assumed to be implicit, since they have no incidence on the runtime. A coercion to type $\langle I; O; \rightarrow\rangle$ of any mixin module e is implemented by $(e + \langle I; \epsilon\rangle)_{:\mathit{dom}(O)}$. 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 $\mathit{block}(e, n)$ au calcul $\lambda_\circ$ t.q.

- $\mathit{block}(\square, n)$ est un lift context et un strict context,
- $\mathit{Size}(\mathit{block}(e, n)) = n$,
- $\mathit{block}(v, n) \longrightarrow v$, pourvu que $\mathit{Size}(v) = n$,
- $[\![\mathit{block}(e, n)]\!] = [\![e]\!]$, mais on perd la compltude, 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'evaluation correct etant donne un graphe et les formes serait une bonne idee. 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 $\lambda_\circ$. De toute facon, pour $\lambda_\circ$, faut la refaire, a cause de ces regles justement.

Headers Ajouter des headers, ca 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 bibtex entry for this before giving the thesis to the referees" or "please verify this information ...". Insrer au premier endroit ou c'est utilis la notation $\perp$ pour les ensembles disjoints, la notation $|\cdot|$ 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 plutt la syntaxe OCaml que la syntaxe SML. Substitution $\{x \mapsto y\}$ signifie que x remplace y. Meta-galits : $=_{\text{def}}$ 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 suppos qu'il y a des types produits, trouver et le dire. Faire plusieurs passes pour vrifier que ces conventions sont respectes. Remplacer les `\\ \noindent` par `\linebreak`. Priorits: dcrire au dbut les priorits implicites, notamment la slection . 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 implementations. 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 reduction 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 sudies 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 gluing 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 astract 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 abstraction facilities.

This thesis examines an atlernative, 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 characterization 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:

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

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

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

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

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

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

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

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

```
module M = struct
  let rec g z = ... f ...
```

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

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

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

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

We refer to [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 let rec construct of ML only allows to bind syntactic functions (or constructed values in the case of OCaml). Therefore, in the case of more complex dependencies between the definitions of a mixin module, instantiation would lead to nested let and let rec bindings. In order to avoid this complication, our calculus features a slightly more powerful let rec than that of ML, which is reminiscent of monadic recursive bindings [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

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

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

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

1.3.3 Typing issues

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

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

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

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

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

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

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:

```
module A =
  struct
    let f x = x
    let x = f 0
  end
```

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 = struct let x = 0 end`, the functor `F` generates an empty module, thus not extending `X` at all.
On the contrary, when composed with a mixin module `X = import export define x = 0 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 designing 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:	$e ::= \mathsf{module}\ \mathit{binding}_1, \ldots, \mathit{binding}_n\ \mathsf{end}$	Basic module
	$\mid e_1 \,\|\, e_2$	Composition
	$\mid e_1 \leftarrow e_2$	Override
	$\mid e\ \mathsf{freeze}\ X$	Freezing
	$\mid e\ \mathsf{freeze\_all\_except}\{X_1 \ldots X_n\}$	Complementary freezing
	$\mid e \setminus X$	Deletion
	$\mid e\ \pi\ X_1 \ldots X_n$	Projection
	$\mid e\ \mathsf{hide}\ X$	Hiding
	$\mid e\ \mathsf{show}\{X_1 \ldots X_n\}$	Show
	$\mid e[X_1 \mapsto X_2]$	Renaming
	$\mid e\ \mathsf{copy}\ X\ \mathsf{as}\ Y$	Copy as
Instance:	$i ::= \mathsf{instantiate}\ e$	
Definition sequence:	$\mathit{binding} ::= X \mid \mathit{def}$	Binding
	$\mathit{def} ::= X = e \mid X = E$	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 ($e_2\ \mathsf{hide}\ X$). 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 `Border` be a mixin defining the functions `display` and `display_border`, and declaring only the missing function `display_body`. The function `display` successively calls `display_border` and `display_body`.

```
Border = module
  display_body,
  display () =
    display_border () ;
    display_body ()
end
```

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 epxressive 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 level recursive scope 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.

Core identifiers	Module identifiers	Type identifiers	Constructors
$x \in \mathit{Vars}$	$s \in \mathit{MVars}$	$t \in \mathit{TVars}$	
$\mathbf{x} \in \mathit{Names}$	$\mathbf{s} \in \mathit{MNames}$	$\mathbf{t} \in \mathit{TNames}$	$\mathbf{c} \in \mathit{CNames}$

$$\begin{array}{lrl}
\text{Path:} & p & ::= \epsilon \mid s \mid p.\mathbf{s} \\
\\
\text{Module:} & M_t & ::= \mathsf{struct}\, M\, \mathsf{end} \mid \mathsf{functor(structure}\, s_{\mathbf{s}} : S_t)M_t \mid (M_t : S_t) \\
 & & \quad \mid p \mid p(p) \mid M\zeta(M_b)M \mid M_t \otimes M_t \mid \mathsf{clos}(M_t) \\
\\
\text{Mixin body:} & M_b & ::= \mathsf{val}\, f_{\mathbf{f}} = \lambda x.E \mid \mathsf{type}\, t_{\mathbf{t}} \mid t\, \mathsf{is}\, \Phi \mid M_b; M_b \mid \epsilon \\
\text{Module body:} & M & ::= \mathsf{structure}\, s_{\mathbf{s}} = M_t \mid \mathsf{type}\, t_{\mathbf{t}} \mid t = \tau \mid t\, \mathsf{is}\, \Phi \mid M; M \mid D \\
\text{Value definition:} & D & ::= \mathsf{fun}\, f^1_{\mathbf{f}} = \lambda x_1.E_1 \mid\mid \ldots \mid\mid f^2_{\mathbf{f}} = \lambda x_2.E_2 \\
 & & \quad \mid \mathsf{val}\, x_{\mathbf{x}} = E \mid D; D \mid \epsilon \\
\\
\text{Core:} & E & ::= \bot \mid x \mid p.\mathbf{x} \mid p.\mathbf{c}(E_1 \ldots E_n) \mid EE \mid \lambda x.E \\
 & & \quad \mid \mathsf{let}\, D\, \mathsf{in}\, E \mid \mathsf{case}\, E\, \mathsf{of}\, R_1\, '|' \ldots '|'\, R_n \mid \mathsf{inner} \\
\text{Matching:} & R & ::= P \Rightarrow E \\
\text{Pattern:} & P & ::= x \mid p.\mathbf{c}(x_1 \ldots x_n) \\
\\
\text{Module type:} & S_t & ::= \mathsf{sig}\, S\, \mathsf{end} \mid \mathsf{funsig(structure}\, s_{\mathbf{s}} : S_t)S_t \mid S\zeta(S)S \\
\text{Signature:} & S & ::= \mathsf{type}\, t_{\mathbf{t}} \mid t = \tau \mid t\, \mathsf{is}\, \Phi \mid \mathsf{val}\, x_{\mathbf{x}} : \tau \\
 & & \quad \mid \mathsf{structure}\, s_{\mathbf{s}} : S_t \mid S; S \mid \epsilon \\
\text{Core type:} & \tau & ::= t \mid p.\mathbf{t} \mid \tau_1 \to \tau_2 \\
\text{Data type definition:} & \Phi & ::= \mathbf{c}(\tau_1 \ldots \tau_n) \mid \Phi \cup \Phi
\end{array}$$

Figure 2.2: Syntax for *DS*

2.2 Duggan and Sourelis' mixin modules

In [31, 32], Duggan and Sourelis 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 $\mathbf{x}$, $\mathbf{s}$, $\mathbf{t}$ and variables x, s, t, respectively for the core language, for modules, and for types. A basic mixin module $A =_{\text{def}} M_1\zeta(M_b)M_2$ 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_2 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 $\otimes$. 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 $\lambda x.E\{\mathsf{inner} \mapsto \lambda y.E_2\}$
- Two data type definitions Φ_1 and Φ_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 $M_1^1\zeta(M_b^1)M_2^1$ and $M_1^2\zeta(M_b^2)M_2^2$ is

$$(M_1^2; M_1^1)\zeta(M_b^2 \otimes M_b^1)(M_2^2; M_2^1),$$

where $\otimes$ 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 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 $\mathsf{struct}\, M\, \mathsf{end}$, functors $\mathsf{functor}(\mathsf{structure}\, s_{\mathbf{s}} : S_t)M_t$ (where S_t is a mixin module type). The core language is a toy functional language with pattern matching, and two special constructs:

- the $\bot$ "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.

$$\begin{array}{lll} \mathit{Num} =_{\mathrm{def}} \quad \zeta(& \mathsf{type}\, t_{\mathbf{term}}; & t\, \mathsf{is}\, \mathbf{Const}(\mathit{int}) \\ & \mathsf{type}\, v_{\mathbf{value}}; & v\, \mathsf{is}\, \mathbf{Num}(\mathit{int}) \\ & \mathsf{type}\, e_{\mathbf{env}}; & e = \mathit{string} \rightarrow v \\ & \mathit{eval}_{\mathbf{eval}} = & \lambda x.\lambda \mathit{env}.\, \mathsf{case}\, x\, \mathsf{of} \\ & & '|'\ \mathbf{Const}(i) \Rightarrow \mathbf{Num}(i) \\ & & '|'\ x \Rightarrow (\mathsf{inner}\ x\ \mathit{env})\big) \end{array}$$

The *Num* mixin module defines the **Const** constructor for the type **term** of terms, and the part of the **eval** evaluation function which evaluates it to the corresponding **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*:

$$
\begin{array}{lllll}
\mathit{Func} =_{\mathrm{def}} & \big(& \multicolumn{3}{l}{\mathsf{fun}\ \mathit{bind}_{\mathbf{bind}} = \lambda x.\lambda v.\lambda \mathit{env}.\lambda y.\,\mathsf{if}\ x = y\ \mathsf{then}\ v\ \mathsf{else}(\mathit{env}\ y)} \\
& \zeta\big(& \mathsf{type}\ t_{\mathbf{term}}; & t\ \mathsf{is}\ \ \mathbf{Var} & (\mathit{string}) \\
& & & \quad \cup\,\mathbf{Abs} & (\mathit{string}, t) \\
& & & \quad \cup\,\mathbf{App} & (t, t) \\
& & \mathsf{type}\ v_{\mathbf{value}}; & v\ \mathsf{is}\,\mathbf{Clos}(t, e) & \\
& & \mathsf{type}\ e_{\mathbf{env}}; & e = \mathit{string} \rightarrow v & \\
& & \mathit{eval}_{\mathbf{eval}} = & \multicolumn{2}{l}{\lambda x.\lambda \mathit{env}.\,\mathsf{case}\ x\ \mathsf{of}} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{Var}(s) \Rightarrow (\mathit{env}\ s)} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{Abs}(s, \mathit{term}) \Rightarrow \mathbf{Clos}(\mathbf{Abs}(s, \mathit{term}), \mathit{env})} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{App}(f, \mathit{term}) \Rightarrow} \\
& & & \multicolumn{2}{l}{\quad (\mathsf{case}\ \mathit{eval}\ f\ \mathit{env}\ \mathsf{of}} \\
& & & \multicolumn{2}{l}{\quad \text{'}|\text{'}\ \mathbf{Clos}(\mathbf{Abs}(s, \mathit{fbody}), \mathit{env}') \Rightarrow} \\
& & & \multicolumn{2}{l}{\qquad \mathit{eval}\ \mathit{fbody}\ (\mathit{bind}\ s\ (\mathit{eval}\ \mathit{term}\ \mathit{env})\ \mathit{env}')} \\
& & & \multicolumn{2}{l}{\quad \text{'}|\text{'}\ x \Rightarrow \mathit{raise}\ \mathit{Error})} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ x \Rightarrow (\mathsf{inner}\ x\ \mathit{env})\big)\big)}
\end{array}
$$

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

$$
\begin{array}{lllll}
\mathit{Interp} =_{\mathrm{def}} & \big(& \multicolumn{3}{l}{\mathsf{fun}\ \mathit{bind}_{\mathbf{bind}} = \lambda x.\lambda v.\lambda \mathit{env}.\lambda y.\,\mathsf{if}\ x = y\ \mathsf{then}\ v\ \mathsf{else}(\mathit{env}\ y)} \\
& \zeta\big(& \mathsf{type}\ t_{\mathbf{term}}; & t\ \mathsf{is}\ \ \mathbf{Var} & (\mathit{string}) \\
& & & \quad \cup\,\mathbf{Abs} & (\mathit{string}, t) \\
& & & \quad \cup\,\mathbf{App} & (t, t) \\
& & & \quad \cup\,\mathbf{Const}(\mathit{int}) & \\
& & \mathsf{type}\ v_{\mathbf{value}}; & v\ \mathsf{is}\ \ \mathbf{Clos}\ \ (t, e) & \\
& & & \quad \cup\mathbf{Num}\ \ (\mathit{int}) & \\
& & \mathsf{type}\ e_{\mathbf{env}}; & e = \mathit{string} \rightarrow v & \\
& & \mathit{eval}_{\mathbf{eval}} = & \multicolumn{2}{l}{\lambda x.\lambda \mathit{env}.\,\mathsf{case}\ x\ \mathsf{of}} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{Const}(i) \Rightarrow \mathbf{Num}(i)} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{Var}(s) \Rightarrow (\mathit{env}\ s)} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{Abs}(s, \mathit{term}) \Rightarrow \mathbf{Clos}(\mathbf{Abs}(s, \mathit{term}), \mathit{env})} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ \mathbf{App}(f, \mathit{term}) \Rightarrow} \\
& & & \multicolumn{2}{l}{\quad (\mathsf{case}\ \mathit{eval}\ f\ \mathit{env}\ \mathsf{of}} \\
& & & \multicolumn{2}{l}{\quad \text{'}|\text{'}\ \mathbf{Clos}(\mathbf{Abs}(s, \mathit{fbody}), \mathit{env}') \Rightarrow} \\
& & & \multicolumn{2}{l}{\qquad \mathit{eval}\ \mathit{fbody}\ (\mathit{bind}\ s\ (\mathit{eval}\ \mathit{term}\ \mathit{env})\ \mathit{env}')} \\
& & & \multicolumn{2}{l}{\quad \text{'}|\text{'}\ x \Rightarrow \mathit{raise}\ \mathit{Error})} \\
& & & \multicolumn{2}{l}{\text{'}|\text{'}\ x \Rightarrow (\mathsf{inner}\ x\ \mathit{env})\big)\big)}
\end{array}
$$

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 $f_{\mathbf{f}} =$

$\mathsf{fun}\, x.(\mathsf{inner}\, x)$ in the body of a mixin module s_1 has the same effect as abstracting over $f_{\mathbf{f}}$. The advantage is that the mixin module s_2 providing the definition for $f_{\mathbf{f}}$ could perfectly have abstracted over another definition $g_{\mathbf{g}}$, 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 $\tau^1_1 \dots \tau^1_{n_1}$ and $\tau^2_1 \dots \tau^2_{n_2}$, are merged. The result is a type constructor expecting the list of types $\tau^1_1 \dots \tau^1_{n_1}; \tau^2_1 \dots \tau^2_{n_2}$. 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 $B =_{\mathrm{def}} \zeta(\mathsf{fun}\, f_{\mathbf{f}} = E)$, obtaining $B \otimes A$. The definition of f in $B \otimes A$ is $E\{\mathsf{inner} \mapsto E_0\}$, 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 int for example. The equational theory of types does not contain the equality $t = int$, 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_t 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 *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)

```
(compound-unit
(import ...)
(export ...)
(link (A)
      (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.

```
(unit
  (import error)
  (export new insert delete)
  (define new ···)
  (define insert ···)
  (define delete ···))
```

Nothing special here, it resembles a function. But now, let `GUI` be:

$$
\begin{array}{rcl}
v & ::= & \mathit{unit} \mid c \mid \mathsf{fn}\, x \Rightarrow e \\
e & ::= & \mathit{compound\text{-}expr} \mid \mathit{invoke\text{-}expr} \mid \mathit{letrec\text{-}expr} \mid e; e \mid x \mid ee \mid v \\
\mathit{unit\text{-}expr} & ::= & \mathsf{unit} \quad \mathsf{import}\, \mathit{variable\text{-}mapping}^* \\
 & & \phantom{\mathsf{unit}} \quad \mathsf{export}\, \mathit{variable\text{-}mapping}^* \\
 & & \phantom{\mathsf{unit}} \quad \mathit{definitions}\ e \\
\mathit{compound\text{-}expr} & ::= & \mathsf{compound} \quad \mathsf{import}\, y^* \\
 & & \phantom{\mathsf{compound}} \quad \mathsf{export}\, y^* \\
 & & \phantom{\mathsf{compound}} \quad \mathsf{link}\, e\ \mathit{link}\, \mathsf{and}\, e\ \mathit{link} \\
\mathit{invoke\text{-}expr} & ::= & \mathsf{invoke}\, e\, \mathsf{with}\, \mathit{value\text{-}invoke\text{-}link}^* \\
\mathit{letrec\text{-}expr} & ::= & \mathsf{let\ rec}\ \mathit{definitions}\ \mathsf{in}\ e \\
\mathit{definitions} & ::= & \mathit{value\text{-}defn}^* \\
\mathit{value\text{-}defn} & ::= & \mathsf{val}\, x = v \\
\mathit{link} & ::= & \mathsf{with}\, y^*\ \mathsf{provides}\, y^* \\
\mathit{variable\text{-}mapping} & ::= & y = x \\
\mathit{value\text{-}invoke\text{-}link} & ::= & y = e \\
x & ::= & \text{variable} \\
y & ::= & \text{linking variable} \\
c & ::= & \text{primitive constant}
\end{array}
$$

Figure 2.3: Syntax for $\mathbf{Unit}_{\mathrm{d}}$

```
(unit
  (import insert)
  (export open error)
  (define open ···)
  (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 `PROGRAM`. As we have both `DB` depending on `GUI` through `error` and *vice versa* through `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 `invoke` form, as in `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.

2.3.2 Theory

In his thesis [35], Flatt formalizes a theory of units, in three calculi. $\mathbf{Unit}_{\mathrm{d}}$, the first unit calculus more or less models the behavior of MzScheme. The next two ones ($\mathbf{Unit}_{\mathrm{c}}$ and $\mathbf{Unit}_{\mathrm{e}}$) successively add constructed types and type abbreviations to $\mathbf{Unit}_{\mathrm{d}}$.

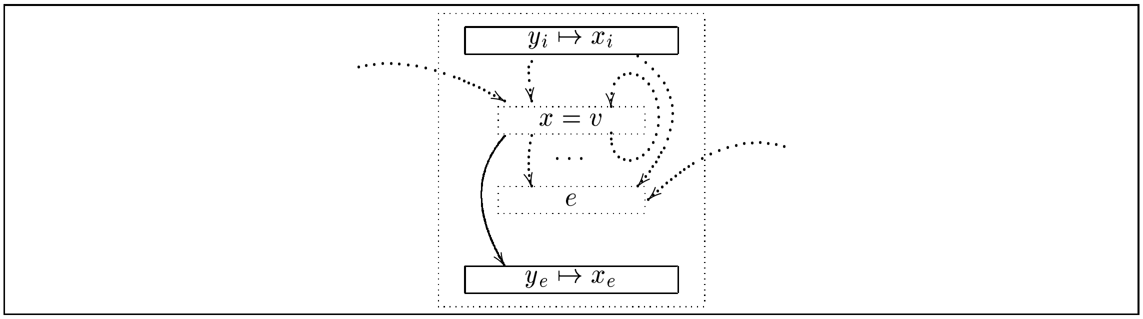


Figure 2.4: A basic unit and its syntactic layers

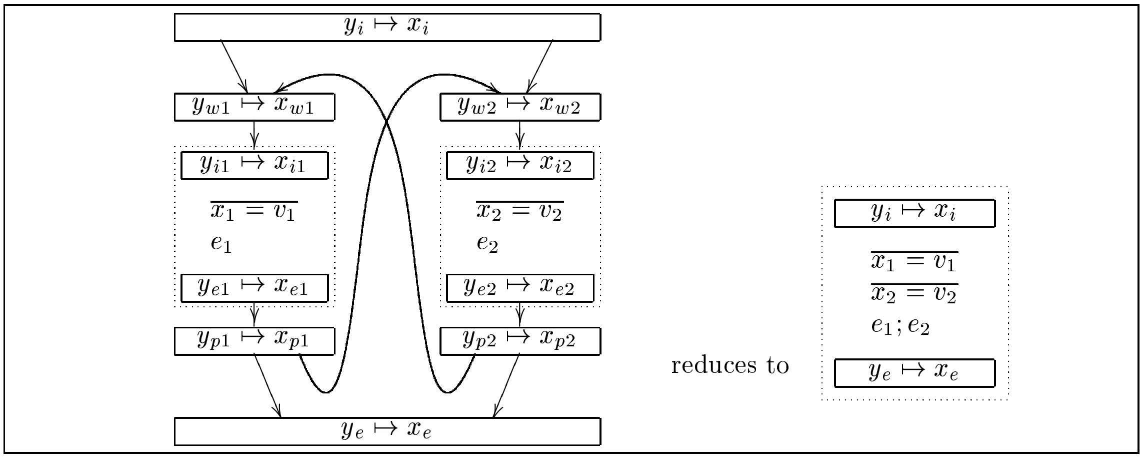


Figure 2.5: A compound unit and its reduction

The Unit$_\mathbf{d}$ calculus The syntax for **Unit**$_\mathrm{d}$ is defined in figure 2.3. A unit is a quadruple of a list of imports, of the shape $y_1 = x_1 \dots y_n = x_n$, a list of exports, of the same shape, a list of value definitions $x_1 = v_1 \dots x_n = v_n$, 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 composes two units A and B as follows.

compound
  import $\overline{y_i}$
  export $\overline{y_e}$
  link A with $\overline{y_{w1}}$ provides $\overline{y_{e1}}$
  and B with $\overline{y_{w2}}$ provides $\overline{y_{e2}}$

The notation $\overline{y}$ represents a sequence of linking variables. Two intermediate layers of variable mappings are introduced: the import layer $\overline{y_{w1}}\ \overline{y_{w2}}$ and the export layer $\overline{y_{e1}}\ \overline{y_{e2}}$. 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 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.

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 *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 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.

Example 2 *Assume that we are supposed to write a unit which prints ML-like type variables: they are represented as records, but are unnamed, and the unit must therefore choose names* `'a`*,* `'b`*, etc..., and provide a* `reset` *function.*

In OCaml, this is done with the functor of figure 2.6, with an internal reference, which the `print` *function increments, and the* `reset` *function resets. With* $\textbf{Unit}_d$*, 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 possibility can be sketched as follows*

```
let PrintTyVar = fn() ⇒
    let x = ref 'a' in
    unit  import error eq
          export print_tyvar
             ...
```

```
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
      snd (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.fprintf fmt "%s" (string_of_tyvar v)

end
```

Figure 2.6: Printing type variables in OCaml

(`let x = e1 in e2` *is syntactic sugar for* `(fn x ⇒ e2) e1`*.) 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 $\mathbf{Unit}_{\mathrm{d}}$ with types are presented. $\mathbf{Unit}_{\mathrm{c}}$ introduces constructed types in a simplified form. Declarations of the shape

$$\mathbf{type}\ t = x_c^1, x_d^1\ \tau_1\ \mid\ x_c^2, x_d^2\ \tau_2 \diamond x_m$$

are allowed, and should be read as the declaration of a type with two constructors $x_c^1 : \tau_1 \to t$ and $x_c^2 : \tau_2 \to t$, two destructors $x_d^1 : t \to \tau_1$ and $x_d^2 : t \to \tau_2$, and a filter x_m, which takes a value of type t as argument and returns **true** if it is of the form $(x_c^1\, v)$ and **false** otherwise.

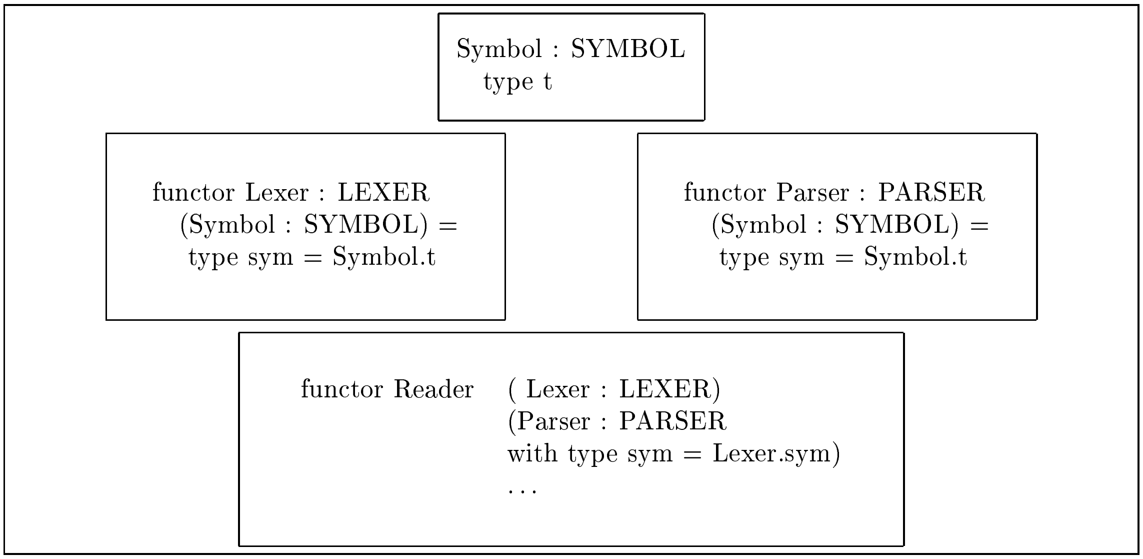
Types may be exported (as abstract types), thanks to type linking variables s. The programmer may decide to export the constructors and destructors for his type, or not. In contrast with the manifest types [51] – translucent sums [40] systems, there is no mechanism for externally selecting components, 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 $x_d^1(x_c^2 e)$ 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 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 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,



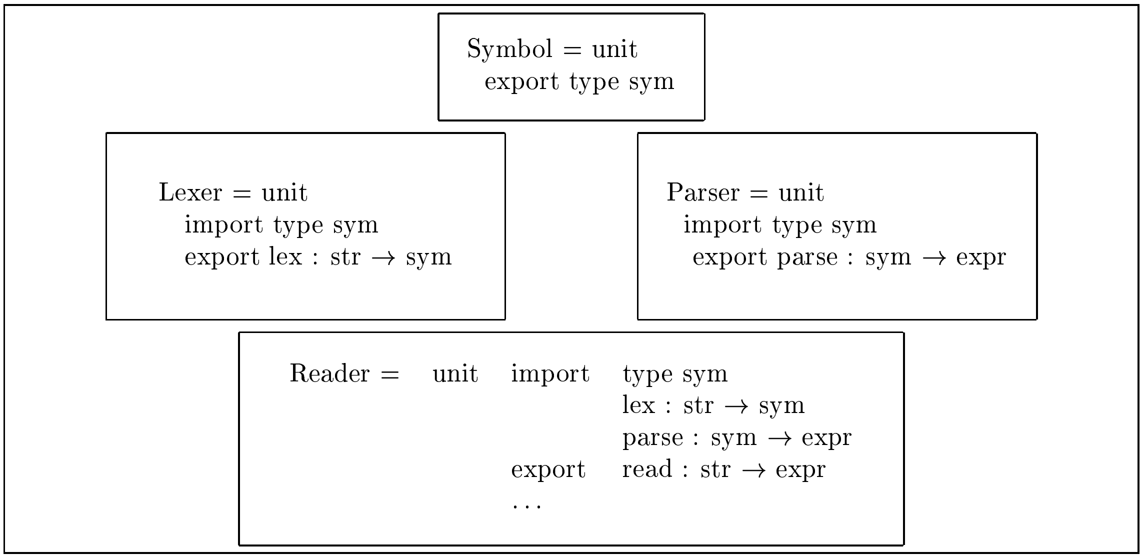
```
Symbol : SYMBOL
   type t

functor Lexer : LEXER
   (Symbol : SYMBOL) =
    type sym = Symbol.t

functor Parser : PARSER
   (Symbol : SYMBOL) =
    type sym = Symbol.t

functor Reader    ( Lexer : LEXER)
                  (Parser : PARSER
                  with type sym = Lexer.sym)
                  . . .
```

Figure 2.7: An example diamond problem, with functors



```
Symbol = unit
  export type sym

Lexer = unit
   import type sym
   export lex : str → sym

Parser = unit
  import type sym
   export parse : sym → expr

Reader =    unit    import    type sym
                              lex : str → sym
                              parse : sym → expr
                    export    read : str → expr
                    . . .
```

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 loosing 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 α-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 ρ in the style of [2], which must cover the whole set of its free variables, and not be recursive ($FV(cod(\rho)) \perp dom(\rho)$).

CMS basic modules are constructed by the $[\iota; o; \rho]$ form. The meta-variable ι ranges over input *assignments*, which are lists of bindings from variables to names, written $x_i \overset{i \in I}{\mapsto} X_i$. 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_is must be different. Output assignments o map names to expressions, and represent the definitions exported by the module. Local assignments ρ map variables to expressions, and are the hidden definitions of the module. The scope of the variables

Expression:	$E ::= x$	Variable
	$\mid C[\rho]$	Core expression
	$\mid [\iota; o; \rho]$	Basic module
	$\mid E_1 + E_2$	Sum
	$\mid {}_{\sigma^\iota \mid}E_{\mid \sigma^o}$	Reduct
	$\mid \mathit{freeze}_{\sigma^f}(E)$	Freeze
	$\mid E.X$	Select
Finite maps:	$\iota ::= x_i \overset{i\in I}{\mapsto} X_i$	Input assignment
	$o ::= X_i \overset{i\in I}{\mapsto} E_i$	Output assignment
	$\rho ::= x_i \overset{i\in I}{\mapsto} E_i$	Local assignment
	$\sigma ::= X_i \overset{i\in I}{\mapsto} Y_i, Y_j^{j\in J}$	Renaming

Figure 2.9: *CMS* syntax

bound by ι and ρ is the whole mixin module. A basic module is well-formed if ι and ρ do not bind any variable in common. Composition $\sigma_1 \circ \sigma_2$ is defined on finite maps, only if $cod(\sigma_2) \subset dom(\sigma_1)$. Union $\sigma_1 + \sigma_2$ is defined on finite maps σ_1 and σ_2, provided $dom(\sigma_1) \perp dom(\sigma_2)$.

Module operators include composition, here called the sum, which links two modules together. The reduct operator ${}_{\sigma^\iota \mid}E_{\mid \sigma^o}$, is roughly a powerful renaming operator, but not only, since it expresses definition hiding. Here σ^ι and σ^o are renaming, which syntactically are pairs of an 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 α-convertible in basic modules, and we will consider expressions modulo α-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 $\longrightarrow$ 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 $\longrightarrow$ of *CMS* includes the transitive closure of the reduction on core expressions $\longrightarrow_C$, 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, explicitely bound in the surrounding substitution ρ. 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 ρ_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 $o_1 + o_2$ implies that o_1 and o_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:

$$\frac{\begin{array}{c}(dom(\iota_1) \cup dom(\rho_1)) \perp (FV([\iota_2; o_2; \rho_2])) \\ (dom(\iota_2) \cup dom(\rho_2)) \perp (FV([\iota_1; o_1 + o; \rho_1])) \qquad dom(o) \subset dom(o_2)\end{array}}{([\iota_1; o_1 + o; \rho_1] \leftarrow [\iota_2; o_2; \rho_2]) \longrightarrow [\iota_1 + \iota_2; o_1 + o_2; \rho_1 + \rho_2]} \quad (\text{AZ-Override})$$

Rule AZ-Reduct describes the action of the reduct operator ${}_{\sigma^\iota|}E_{|\sigma^o}$. In fact, it could be divided into two operators, one for reducing input ${}_{\sigma^\iota|}E$, and one for reducing outputs $E_{|\sigma^o}$. Both actions are similar though. Each of them bases on a renaming σ^ι and σ^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 σ^ι includes a binding $X \mapsto Y$, and ι includes a binding $x \mapsto X$. Then, the composition of ι and σ^ι has a binding $x \mapsto Y$. If the renaming has unused names, or if $dom(\sigma^\iota)$ contains names not in $cod(\iota)$, 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

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

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 $[; o; \rho]$, with $\rho = x_i \overset{i \in I}{\mapsto} E_i$. 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 occurence of $x_i \in dom(\rho)$ in E is replaced with a kind of closure: the local definitions ρ 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 $E \longrightarrow^* E_1$ and $E \longrightarrow^* E_2$, then there exists E' such that $E_1 \longrightarrow^* E'$ and $E_2 \longrightarrow^* E'$.*

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 $[\Sigma^\iota; \Sigma^o]$, where Σ^ι and Σ^o are *signatures*, i.e. finite sets of pairs of a name and a type. Σ^ι is the input signature, representing the requirements put on inputs, while Σ^o is the output signature, declaring the capabilities offered by the output definitions. Typing judgments are parameterized by the corresponding judgments

$$\frac{C \longrightarrow_C^+ C'}{C[\rho] \longrightarrow C'[\rho]} \quad (\text{AZ-}\textsc{Core}) \qquad C[x \mapsto C'[\rho_1], \rho_2] \longrightarrow C\{x \mapsto C'\}[\rho_1, \rho_2] \quad (\text{AZ-}\textsc{Sub})$$

$$\frac{E_1 = [\iota_1; o_1; \rho_1] \qquad E_2 = [\iota_2; o_2; \rho_2] \qquad BV(E_1) \perp FV(E_2) \qquad BV(E_2) \perp FV(E_1)}{E_1 + E_2 \longrightarrow [\iota_1 + \iota_2; o_1 + o_2; \rho_1 + \rho_2]} \quad (\text{AZ-}\textsc{Sum})$$

$${}_{\sigma^\iota|}[\iota; o; \rho]_{|\sigma^o} \longrightarrow [\sigma^\iota \circ \iota; o \circ \sigma^o; \rho] \quad (\text{AZ-}\textsc{Reduct})$$

$$\frac{\mathit{cod}(\iota_2) \perp \mathit{dom}(\sigma^f)}{\mathit{freeze}_{\sigma^f}([\iota_1 + \iota_2; o; \rho]) \longrightarrow [\iota_2; o; \rho + o \circ \sigma^f \circ \iota_1]} \quad (\text{AZ-}\textsc{Freeze})$$

$$[; o; x_i \overset{i \in I}{\mapsto} E_i].X \longrightarrow o(X)\{x_j \overset{j \in I}{\mapsto} [; Y \mapsto E_j; x_i \overset{i \in I}{\mapsto} E_i].Y\} \quad (\text{AZ-}\textsc{Select})$$

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 $\iota = x_i \overset{i \in I}{\mapsto} X_i$ and those of the local definitions $\rho = x_k \overset{k \in K}{\mapsto} E_k$, say $x_i : \tau_i^{i \in I \cup K}$. With these types, the local definitions can be checked to have the expected types $\tau_k^{k \in K}$, and the exported definitions can be typed $\tau_j^{j \in J}$. 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 $[\Sigma^\iota; \Sigma^o]$, guesses two signatures $\Sigma^{\iota\prime}$ and $\Sigma^{o\prime}$ such that the input and output renamings respectively map Σ^ι to $\Sigma^{\iota\prime}$ and $\Sigma^{o\prime}$ to Σ^o, preserving types, as witnessed by the side-conditions $\sigma^\iota : \Sigma^\iota \to \Sigma^{\iota\prime}$ and $\sigma^o : \Sigma^{\iota\prime} \to \Sigma^o$. 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 specfication, preserving 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.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\text{AZ-T-V}\textsc{ar}) \qquad \frac{x_i : c\tau_i^{i \in I} \vdash_C C : c\tau \qquad \forall i \in I, \Gamma \vdash E_i : c\tau_i}{\Gamma \vdash C[x_i \overset{i \in I}{\mapsto} E_i] : c\tau}\ (\text{AZ-T-C}\textsc{ore})$$

$$\frac{\vdash [X_i : \tau_i^{i \in I}; X_j : \tau_j^{j \in J}] \qquad \forall j \in J \cup K, \Gamma + x_i : \tau_i^{i \in I \cup K} \vdash E_j : \tau_j}{\Gamma \vdash [x_i \overset{i \in I}{\mapsto} X_i; X_j \overset{j \in J}{\mapsto} E_j; x_k \overset{k \in K}{\mapsto} E_k] : [X_i : \tau_i^{i \in I}; X_j : \tau_j^{j \in J}]}\ (\text{AZ-T-B}\textsc{asic})$$

$$\frac{\begin{array}{c}\vdash [\Sigma^{\iota}{}_1 + \Sigma^{\iota}{}_2; \Sigma^{o}{}_1 + \Sigma^{o}{}_2] \\ \Sigma^{o}{}_1 \perp \Sigma^{o}{}_2 \qquad \Gamma \vdash E_1 : [\Sigma^{\iota}{}_1; \Sigma^{o}{}_1] \qquad \Gamma \vdash E_2 : [\Sigma^{\iota}{}_2; \Sigma^{o}{}_2]\end{array}}{\Gamma \vdash E_1 + E_2 : [\Sigma^{\iota}{}_1 + \Sigma^{\iota}{}_2; \Sigma^{o}{}_1 + \Sigma^{o}{}_2]}\ (\text{AZ-T-S}\textsc{um})$$

$$\frac{\Gamma \vdash E : [\Sigma^{\iota}; \Sigma^{o}] \qquad \sigma^{\iota} : \Sigma^{\iota} \to \Sigma^{\iota\prime} \qquad \sigma^{o} : \Sigma^{\iota\prime} \to \Sigma^{o}}{\Gamma \vdash {}_{\sigma^{\iota}|}E_{|\sigma^{o}} : [\Sigma^{\iota\prime}; \Sigma^{o\prime}]}\ (\text{AZ-T-R}\textsc{educt})$$

$$\frac{\Gamma \vdash E : [\Sigma^{f} + \Sigma^{\iota}; \Sigma^{o}] \qquad \sigma^{f} : \Sigma^{f} \to \Sigma^{o} \qquad \Sigma^{f} \perp \Sigma^{\iota}}{\Gamma \vdash \mathit{freeze}_{\sigma^{f}}(E) : [\Sigma^{\iota}; \Sigma^{o}]}\ (\text{AZ-T-F}\textsc{reeze})$$

$$\frac{\Gamma \vdash E : [; X_i : \tau_i^{i \in I}] \qquad k \in I}{\Gamma \vdash E.X_k : \tau_k}\ (\text{AZ-T-S}\textsc{elect})$$

Figure 2.11: Typing *CMS*

2.4.3 Expressiveness and inconvenients

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 $\lambda x.e$, as $[x \mapsto \mathit{ARG}; \mathit{RES} \mapsto [\![e]\!];]$ (where $[\![\cdot]\!]$ denotes the encoding function). Function application $e_1 e_2$ can then be expressed as

$$(\mathit{freeze}_{\mathit{ARG} \mapsto \mathit{ARG}}([\![e_1]\!] + [; \mathit{ARG} \mapsto [\![e_2]\!];])).\mathit{RES}.$$

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 $e.l$ is then encoded as $(\mathit{freeze}_{\mathit{SELF} \mapsto \mathit{SELF}}([\![e]\!] + [; \mathit{SELF} \mapsto [\![e]\!];])).l$.

***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 [?], 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)

```
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 Sourelis' $\mathsf{clos}()$ operator, and of Flatt's instantiation operator. In the following, we call this operation *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.

[1] Elena Zucca, personal communication, 2003

$$\begin{array}{rll}
x, y, z & \in \mathit{Vars} & \text{Variable} \\
X, Y, Z & \in \mathit{Names} & \text{Name} \\
L & ::= X \mid \_ & \text{Label} \\
B & ::= e \mid \bullet & \text{Body} \\
d & ::= L \triangleright x = B & \text{Definition} \\
b & ::= d_1 \dots d_n & \text{Binding } (n \geq 0) \\
e, f & ::= x & \text{Variable} \\
 & \mid \langle b \rangle & \text{Mixin module} \\
 & \mid e \oplus f & \text{Linking} \\
 & \mid e_{:-X} & \text{Hiding} \\
 & \mid e.X & \text{Selection} \\
 & \mid \mathsf{let\ rec}\ b\ \mathsf{in}\ e & \text{Let rec}
\end{array}$$

Figure 2.12: Syntax of the **m**-calculus

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-calulus

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 $x, y, z \in \mathit{Vars}$ are distinguished from names $X, Y, Z \in \mathit{Names}$. A *mixin module* $\langle b \rangle$ in **m** consists of *binding* b. A binding is a list $\langle d_1 \dots d_n \rangle$ of definitions d_i. A *definition* $d = (L \triangleright x = B)$ 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 $e \oplus f$, the *hiding* of a name in an expression $e_{:-X}$, 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 $\mathsf{let\ rec}$ 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 $b_{\backslash\mathcal{N}}$ 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 $\mathsf{let\ rec}$ are α-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 $\mathbf{m}$ is defined as the least contextually closed relation respecting the rules in figure 2.13.

The main and most complicated rule is the WV-Link for linking two mixin modules $\langle b\rangle$ and $\langle b'\rangle$. First, the definitions bound by the same names in the two bindings are isolated, the other ones being copied straightfowardly into the result mixin module. The notation $\mathit{DN}(b)$ denotes the set of names defined by a binding, so $\mathcal{N} = \mathit{DN}(b) \cap \mathit{DN}(b')$ is the set of names defined in common by the two mixin modules. Let those common definitions be $b_{|\mathcal{N}} = (X_1 \triangleright x_1 = B_1 \dots X_n \triangleright x_n = B_n)$ and $b'_{|\mathcal{N}} = (X_1 \triangleright x_1 = B'_1 \dots X_n \triangleright x_n = B'_n)$. Then, for each pair of similarly named definitions, the function *PickBody* choose the non-empty body if any, and otherwise denotes $\bullet$: $\mathit{PickBody}(e, \bullet) = \mathit{PickBody}(\bullet, e) = e$ and $\mathit{PickBody}(\bullet, \bullet) = \bullet$. 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 $L_1 \triangleright x_1 = \mathbb{C}[x_2]$, and x_2 binds another definition $L_2 \triangleright x_2 = e$, it is allowed to copy e into $\mathbb{C}[x_2]$, 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 $\rightarrow_{\langle b\rangle}$ of our binding b. This graph has the variables defined by the binding as nodes, and its edges are built 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 $x_2 \rightarrow_{\langle b\rangle} x_1$. 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 $\mathsf{let\ rec}$.

In $\mathsf{let\ rec}$, however, the values defined in the binding can also be used in the body of the $\mathsf{let\ rec}$, as stated by rule WV-ESubst. The side-condition just ensures that no variable capture occurs, and that the occurence of x in the body of the $\mathsf{let\ rec}$ actually refers to the considered binding of x.

$$\frac{\begin{array}{c}\mathcal{N} = DN(b) \cap DN(b') \qquad b_{|\mathcal{N}} = (X_1 \triangleright x_1 = B_1 \ldots X_n \triangleright x_n = B_n) \\ b'_{|\mathcal{N}} = (X_1 \triangleright x_1 = B'_1 \ldots X_n \triangleright x_n = B'_n) \\ b'' = (X_1 \triangleright x_1 = B''_1 \ldots X_n \triangleright x_n = B''_n) \qquad \forall 1 \leq i \leq n, B''_i = \mathit{PickBody}(B_i, B'_i)\end{array}}{\langle b \rangle \oplus \langle b' \rangle \longrightarrow \langle b_{\setminus\mathcal{N}} + b_{\setminus\mathcal{N}} + b'' \rangle} \quad (\text{WV-Link})$$

$$\frac{\begin{array}{c} b = (L_1 \triangleright x_1 = \mathbb{C}\,[x_2], L_2 \triangleright x_2 = e, b') \\ \mathit{Capt}_{\Box}(\mathbb{C}\,) \perp (\{x_2\} \cup FV(e)) \qquad x_1 \not\twoheadrightarrow^*_{\langle b \rangle} x_2 \end{array}}{b \longrightarrow (L_1 \triangleright x_1 = \mathbb{C}\,[e], L_2 \triangleright x_2 = e, b')} \quad (\text{WV-ISubst})$$

$$\frac{\mathit{Capt}_{\Box}(\mathbb{C}\,) \perp \{x\} \cup FV(b(x))}{\mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ \mathbb{C}\,[x] \longrightarrow \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ \mathbb{C}\,[b(x)]} \quad (\text{WV-ESubst})$$

$$\frac{DN(b') = \emptyset \qquad DV(b') \perp FV(b) \qquad b' \neq \epsilon}{\langle b + b' \rangle \longrightarrow \langle b \rangle} \quad (\text{WV-GC-Module})$$

$$\frac{DV(b') \perp (FV(b) \cup FV(e)) \qquad b' \neq \epsilon}{\mathsf{let\ rec}\ \ b + b'\ \ \mathsf{in}\ \ e \longrightarrow \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e} \quad (\text{WV-GC-Letrec})$$

$$\mathsf{let\ rec}\ \ \epsilon\ \ \mathsf{in}\ \ e \longrightarrow e \quad (\text{WV-Empty-Letrec})$$

$$\frac{b' \neq \epsilon \qquad DV(b) \perp (DV(b') \cup FV(b'))}{\mathsf{let\ rec}\ \ b'\ \ \mathsf{in}\ \ \langle b \rangle \longrightarrow \langle b' + b \rangle} \quad (\text{WV-Closure}) \qquad \frac{X \notin DN(b)}{\langle b \rangle_{:-X} \longrightarrow \langle b \rangle} \quad (\text{WV-Hide-Absent})$$

$$\langle X \triangleright x = B + b \rangle_{:-X} \longrightarrow \langle \_ \triangleright x = B + b \rangle \quad (\text{WV-Hide-Present})$$

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.

2.5.2 Expressiveness and inconvenients

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 $X^{\mathsf{i}} \triangleright x = \bullet$ and an output definition $X^{\mathsf{o}} \triangleright y = e$. 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 $X \overset{+}{:=} Y$. It is used for renaming output definitions, and applied to an expression e by

$$(e \oplus \langle X \triangleright x = \bullet, Y \triangleright y = x\rangle)_{:-X}.$$

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 $X \overset{-}{:=} Y$ 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. $X \overset{+}{:=} Y$.

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.

$$\begin{array}{l}
\langle X \triangleright x = \bullet, Y \triangleright y = e\rangle[X \overset{-}{:=} Y] \\
= (\langle X \triangleright x = \bullet, Y \triangleright y = e\rangle \oplus \langle Y \triangleright y =, X \triangleright x = y\rangle)_{:-X} \\
\longrightarrow \langle X \triangleright x = y, Y \triangleright y = e\rangle_{:-X} \\
\longrightarrow \langle \_ \triangleright x = y, Y \triangleright y = e\rangle
\end{array}$$

The result mixin module is observationally equivalent to $\langle \_ \triangleright x = e, Y \triangleright y = e\rangle$, 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 $[x \mapsto X : T; X \mapsto e;]$ 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 $[\Sigma^\iota; \Sigma^o]$ can be coerced to the type $[\Sigma^{\iota\prime}; \Sigma^{o\prime}]$, provided the following inclusions hold: $\Sigma^\iota \subset \Sigma^{\iota\prime}$, $\Sigma^{o\prime} \subset \Sigma^o$. The coercion is ${}_{\sigma^\iota|}E_{|\sigma^o}$, where σ^ι and σ^o are the canonical injections from $\mathit{dom}(\Sigma^\iota)$ to $\mathit{dom}(\Sigma^{\iota\prime})$ and from $\mathit{dom}(\Sigma^{o\prime})$ to $\mathit{dom}(\Sigma^o)$, respectively.

Independence with respect to the core language The most obvious semantic difference between both calculi is that *CMS* features second-class mixin modules, explicitely 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 $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else} \ldots$ operator. Let $C =_{\mathrm{def}} \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathit{fact}(x-1)$, and $E_0 =_{\mathrm{def}} C[\mathit{fact} \mapsto \mathit{fact}]$, which injects C into the mixin module language. Then, the mixin module $E_1 =_{\mathrm{def}} [; \mathit{FACT} \mapsto E_0; \mathit{fact} \mapsto E_0]$ 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

$$\begin{array}{l}
(\mathit{fact}\ 0)[\mathit{fact} \mapsto E_1.\mathit{FACT}] \\
\longrightarrow (\mathit{fact}\ 0)[\mathit{fact} \mapsto (E_0\{\mathit{fact} \mapsto E_1.\mathit{FACT}\})] \text{ (by rule AZ-S}\textsc{elect}\text{)} \\
\equiv (\mathit{fact}\ 0)[\mathit{fact} \mapsto (C[\mathit{fact} \mapsto E_1.\mathit{FACT}])] \\
\longrightarrow (\mathit{fact}\ 0)\{\mathit{fact} \mapsto C\}[\mathit{fact} \mapsto E_1.\mathit{FACT}] \text{ (by rule AZ-S}\textsc{ub}\text{)} \\
\equiv (C\ 0)[\mathit{fact} \mapsto E_1.\mathit{FACT}] \\
\longrightarrow 1[\mathit{fact} \mapsto E_1.\mathit{FACT}]
\end{array}$$

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 $E_1.FACT$ 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 **m** way, and rely on the mixin module language to express usual core language features.

From concrete mixin modules to modules In **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 $\langle X \triangleright x = x \oplus x \rangle$ has a dependency problem. Nevertheless, all the common recursive definitions behave well in the **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, **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 **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 **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 $\_ \triangleright x = 1 + 2, \_ \triangleright y = 2 * 1$. 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 $e_1 =_{\mathrm{def}} \langle X \triangleright x = \bullet, Y \triangleright y = x + 1\rangle$ and $e_2 =_{\mathrm{def}} \langle X \triangleright x = 0\rangle$. According to the semantics of $\mathbf{m}$, $e_1 \oplus e_2$ can be either $\langle X \triangleright x = 0, Y \triangleright y = x + 1\rangle$ or $\langle Y \triangleright y = x + 1, X \triangleright x = 0\rangle$, 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 $\langle Y \triangleright y = x + 1, X \triangleright x = 0\rangle$, 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) $\{X_1 = e_1 \dots X_n = e_n\}$, and record selection $e.X$, which are standard.

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

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

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

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

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

Figure 3.1: Syntax of *MM*

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

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

Figure 3.2: Meta-variables and notations

$$
\begin{array}{lrl}
\text{Value:} & v & ::= x \mid \{s_v\} \\
 & & \mid \langle X_1 \triangleright x_1 \dots X_n \triangleright x_n ; d_1 \dots d_n \rangle \\
\text{Answer:} & a & ::= v \mid \mathsf{let\ rec}\ \ b_v\ \ \mathsf{in}\ \ v \\
\text{Value sequence:} & s_v & ::= X_1 = v_1 \dots X_1 = v_1 \\
 & b_v & ::= x_1 = v_1 \dots x_n = v_n
\end{array}
$$

Figure 3.3: Values in *MM*

$$
\langle \iota_1 ; o_1 \rangle \Bumpeq \langle \iota_2 ; o_2 \rangle \text{ means } \left\{ \begin{array}{l} \langle \iota_1 ; o_1 \rangle \bumpeq \langle \iota_2 ; o_2 \rangle \text{ and} \\ \langle \iota_2 ; o_2 \rangle \bumpeq \langle \iota_1 ; o_1 \rangle . \end{array} \right.
$$

$$
\begin{array}{l}
\langle \iota_1 ; o_1 \rangle \bumpeq \langle \iota_2 ; o_2 \rangle \text{ means that for all } (L \triangleright x) \in \mathit{dom}(\langle \iota_1 ; o_1 \rangle), \\
x \in \mathit{FV}(o_2) \cup \mathit{Variables}(\langle \iota_2 ; o_2 \rangle) \Rightarrow (L \triangleright x) \in \mathit{dom}(\langle \iota_2 ; o_2 \rangle) \text{ and } L \in \mathit{Names} .
\end{array}
$$

Figure 3.4: Definition of $\Bumpeq$

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

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

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

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

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

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

$$\langle \iota; o \rangle_{:X_1 \ldots X_n} \leadsto_c \langle \iota; Show(o, \{X_1 \ldots X_n\}) \rangle \quad (\textsc{Show})$$

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

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

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

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

$$\frac{\langle \iota_1; o_1 \rangle \Bumpeq \langle \iota_2; o_2 \rangle \qquad Names(o_1) \perp Names(o_2)}{\langle \iota_1; o_1 \rangle + \langle \iota_2; o_2 \rangle \leadsto_c \langle (\iota_1 \cup \iota_2) \setminus Input(o_1, o_2); o_1, o_2 \rangle} \quad (\textsc{Sum})$$

$$\frac{Bind(\overline{o}) \text{ is correct}}{\textsf{close}\langle \emptyset; o \rangle \leadsto_c \textsf{let rec}\, Bind(\overline{o})\, \textsf{in}\, Record(\overline{o})} \quad (\textsc{Close})$$

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 $P_1 \dots P_n$, $P_1 \perp \dots \perp P_n$ means that the P_i's are pairwise disjoint.

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

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

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

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

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

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

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

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

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 $\_$ 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 ι and o with the new ones. Formally, for $\mathcal{N} \subset \mathit{Names}$, we define $r_{\mathcal{N}}$ by $r \cup id_{|\mathcal{N} \backslash dom(r)}$ and for a finite map f with $dom(f) \subset \mathit{Names}$, we define $f\{r\}$ by $f \circ (r_{dom(f)})^{-1}$. The finite map $f\{r\}$ is well-defined provided $r_{dom(f)}$ is injective, which holds as soon as $cod(r) \cap dom(f) \subset dom(r)$ or in other words $dom(f) \perp (cod(r) \setminus dom(r))$. By the side-condition $\mathit{Names}(\langle \iota; o \rangle) \perp (cod(r) \setminus dom(r))$, this is the case for $\iota\{r\}$. (We denote by $\mathit{Names}(\langle \iota; o \rangle)$ the set of names bound by the structure, i.e. $dom(\iota) \cup dom(\mathit{Input}(o))$.) Finally, we define $o\{r\}$ by $o \circ (r_{\mathit{Names}(o)}, id_{\mathit{Vars}})^{-1}$, 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 $g^{-1}(dom(f))$ 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 $r_{\mathit{Names}(\langle \iota; o \rangle)}$ 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 $\langle \iota; o \rangle_{X \succ Y}$ splits it into an input $X \triangleright x$ and a definition $Y[z^*] \triangleright y = e$ (with a fresh y). References to x continue referencing it as an input, but the former definition e remains exported as Y. The operation is different from renaming X to Y or deleting X.

The Sum rule defines the composition of two structures $\langle \iota_1; o_1 \rangle$ and $\langle \iota_2; o_2 \rangle$. The result is a structure $\langle \iota; o \rangle$, defined as follows. ι is the union of ι_1 and ι_2, where names defined in o_1 or o_2 are removed. o is defined as the concatenation of o_1 and o_2. The side condition $\langle \iota_1; o_1 \rangle \Bumpeq \langle \iota_2; o_2 \rangle$ checks that both structures agree on bound variables, and that no free variable is captured. It is defined in figure

Evaluation context:

$$\mathbb{E} \quad ::= \quad \mathbb{F} \mid \mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{F} \mid \mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ e$$

Lift context:

$$\mathbb{L} \quad ::= \quad \{\mathbb{S}\} \mid op[\Box] \mid \Box + e \mid v + \Box$$

Multiple lift context:

$$\mathbb{F} \quad ::= \quad \Box \mid \mathbb{L}[\mathbb{F}]$$

Binding context:

$$\mathbb{B} \quad ::= \quad b_v, x = \Box, b$$

Record context:

$$\mathbb{S} \quad ::= \quad s_v, X = \Box, s$$

Figure 3.6: Evaluation contexts

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

Figure 3.7: Access in evaluation contexts

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

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

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

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

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

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

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

Figure 3.8: Reduction relation

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

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

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

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

In figure 3.7, the capture conditions are formalized with the *Capt* function. $Capt_{\square}(\mathbb{E})$ is the set of bound variables above $\square$ in $\mathbb{E}$. 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 $\overline{o}$, which we define in this section. This operation takes four aspects of its argument into account: its internal dependencies, its fake dependencies, the shapes of its definitions, and its original ordering. Internal dependencies and fake dependencies are considered imperative requirements on the final ordering: if a definition d might call another definition d', then d' must be put before d in the final ordering. The shapes of the definitions are examined in order not to generate a binding with forward references to definitions of unpredictable shape. The original ordering is only used as a hint, in the case where no constraint forces one definition to be put before the other.

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

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 $\chi \in \{\text{☺}, \text{☹}\}$. χ is determined by $Degree(x', e)$, where the *Degree* function is defined for $x \in FV(e)$ by
$$\begin{array}{rcl} Degree(x, \langle \iota; o \rangle) & = & \text{☺} \\ Degree(x, \{s_v\}) & = & \text{☺} \\ Degree(x, e) & = & \text{☹ otherwise.} \end{array}$$
  The *Degree* function is simple, and could be extended as in [12, 45].

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

$$\frac{(L[x_1 \ldots x_n] \triangleright x = e) \in o \qquad (L'[z^*] \triangleright x_i = e') \in o}{x_i \xrightarrow{\frownie}_o x}$$

Figure 3.9: Dependencies in an output

$$\frac{x \xrightarrow{\chi_1}^{+} z \qquad z \xrightarrow{\chi_2} y}{x \xrightarrow{\chi_2}^{+} y} \qquad\qquad \frac{x \xrightarrow{\chi} y}{x \xrightarrow{\chi}^{+} y}$$

Figure 3.10: Transitive closure of $\rightarrow$

- If x' is mentioned in y^*, then an edge from x' to x is drawn, with degree $\frownie$. 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 $\xrightarrow{\frownie}{}_o^+$ gives a conservative approximation of which definition needs the value of which other one in $\mathit{Bind}(\overline{o})$. Reordering o according to $\rightarrow_o$ it is not enough though, because the generated binding might be syntactically incorrect. Indeed, it is forbidden to make forward references to definitions of unpredictable shape inside a binding. Strict forward references to definitions of unpredictable shape already correspond to edges labeled $\frownie$ in $\rightarrow_o$, and are therefore taken into account when reordering according to $\xrightarrow{\frownie}{}_o^+$. Weak forward references to definitions of unpredictable shape correspond to edges labeled $\smiley$ in $\rightarrow_o$, and are therefore not taken into account when reordering according to $\xrightarrow{\frownie}{}_o^+$. Let $\succ_o = \{(x_1, x_2) \mid x_1 \xrightarrow{\smiley} x_2, o(x_1) \notin \mathit{Predictable}\}$. This relation exactly puts weak references to definitions of unpredictable shape in the right order.

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

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 Γ is a finite map from variables to types. We write $\Gamma\langle\Gamma'\rangle$ for the map where the bindings of Γ' have overridden the ones from Γ.

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

$$\begin{tikzcd} N_1 \arrow[r, bend left, "\chi_1"] \arrow[r, bend right, "\chi_2"'] & N_2 & {} \arrow[r, dashed] & {} & N_1 \arrow[r, "\chi_1 \wedge \chi_2"] & N_2 \end{tikzcd} \tag{4.1}$$

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

$$\begin{array}{l} \chi_1 \wedge \chi_2 = ☺ \textit{ if } \chi_1 = \chi_2 = ☺ \\ \chi_1 \wedge \chi_2 = ☹ \textit{ otherwise} \end{array}$$

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 $\langle\iota; o\rangle$. Given an input I (which is arbitrary here, we do not consider type inference or type-checking issues) corresponding to ι, and a type environment Γ_o correponding to o, it checks that the definitions in o indeed have the types mentioned in Γ_o.

$$\begin{array}{lcl} M \in \mathit{Types} & ::= & \{O\} \mid \langle I; O; G\rangle \\ I, O & \in & \mathit{Names} \xrightarrow{\text{Fin}} \mathit{Types} \\ G & \subset_{\text{Fin}} & \{X \xrightarrow{\chi} Y \mid X, Y \in \mathit{Names}, \chi \in \mathit{Degrees}\} \\ \Gamma & \in & \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{Types} \end{array}$$

Figure 4.1: Types

Expressions:

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

$$\frac{I_1 \uplus O_1 \leftrightharpoons I_2 \uplus O_2 \qquad \vdash G_1 \cup G_2 \qquad \Gamma \vdash e_1 : \langle I_1; O_1; G_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2; O_2; G_2 \rangle}{\Gamma \vdash e_1 + e_2 : \langle (I_1 \cup I_2) \setminus (O_1 \cup O_2); O_1 \uplus O_2; G_1 \cup G_2 \rangle} \quad (\text{T-Sum})$$

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : \{O\}}{\Gamma \vdash e.X : O(X)} \quad (\text{T-Rselect})$$

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

Sequences:

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

Figure 4.2: Type system

Lift

Transitive closure through local components

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

Lift

	$\lfloor \rightarrow \rfloor$	$=$	$\rightarrow^{\Box}{}_{\mid \mathit{Names} \times \mathit{Names}}$
Sum	$G_1 + G_2$	$=$	$G_1 \cup G_2$
Freeze	$G \,!\, X$	$=$	$G\{X \leftarrow x\} \cup \{x \overset{\frownie}{\longrightarrow} X\}$ (x not mentioned in G)
Project	$G_{\mid \mathcal{N}}$	$=$	$G_{\mid \mathit{Names} \times \mathcal{N} \times \mathit{Degrees}}$
Delete	$G_{\mid -\mathcal{N}}$	$=$	$G_{\setminus \mathit{Names} \times \mathcal{N} \times \mathit{Degrees}}$
Hide	$G_{:-X_1 \ldots X_n}$	$=$	$G\{X_1 \mapsto x_1 \ldots X_n \mapsto x_n\}$ ($x_1 \ldots x_n$ fresh)
Show	$G_{:X_1 \ldots X_n}$	$=$	$G_{:- \mathit{Targets}(G) \setminus \{X_1 \ldots X_m\}}$
Rename	$G\{r\}$	$=$	$\{(N_1\{r\}, N_2\{r\}, \chi) \mid (N_1, N_2, \chi) \in G\}$
Split	$G_{X \succ Y}$	$=$	$(G \setminus G_{\mid \{. \rightarrow X\}}) \cup \{(Z, Y, \chi) \mid (Z, X, \chi) \in G\}$

Figure 4.3: Graph operations

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

$$\frac{(L_i, x_i) \in \mathit{dom}(\langle \iota; o \rangle) \qquad (L[x_1 \ldots x_n] \triangleright x = e) \in o}{\mathit{Node}(L_i, x_i) \overset{\frownie}{\longrightarrow}_{\langle \iota; o \rangle} \mathit{Node}(L, x)}$$

Figure 4.4: Dependencies in a structure

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

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

Figure 4.5: Well-formed types

The condition $\vdash \rightarrow_{\langle\iota;o\rangle}$ requires some explanation. We saw in section 3.2 that dependencies in an output are represented by its dependency graph $\rightarrow_o$. For structures (which are incomplete outputs), the corresponding notion is the concrete dependency graph. A concrete dependency graph is a graph over nodes. A node N is an element of $\mathit{Nodes} = \mathit{Vars} \cup \mathit{Names}$. 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 *lift* operation $\lfloor\rightarrow_{\langle\iota;o\rangle}\rfloor$, 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 $\xrightarrow{☹}^{+}$ is an ordering on its nodes (written $\vdash\rightarrow$).*

Definition 5 (Well-formed types) *Figure 4.5 defines the sets of well-formed types an inputs (or outputs), as the least relation respecting the rules. A mixin type $\langle I; O; G\rangle$ 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 $\Bumpeq$ 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-Rename, 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 ($cod(r)$). As the contraction rule Rename, it makes use of the $r_{\mathcal{N}}$ function, composed with I and O. The abstract graph is renamed as well.

Given an expression e of type $\langle I; O; G\rangle$, according to rule T-Split, the type of $e_{X \succ Y}$ is as follows. X is added to the input, with the type it had in O. X is renamed to Y in the output. The graph G is modified according to figure 4.3. $G_{|\{. \rightarrow X\}}$ 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 $\mathsf{let\ rec}\ b\ \mathsf{in}\ e$ 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 $Output(b) = Output(x_1 = e_1 \dots x_n = e_n) = (\_[\,] \triangleright x_1 = e_1 \dots \_[\,] \triangleright x_n = e_n)$. We define $\gtrdot_b$ by $\gtrdot_{Output(b)}$. A binding b is said correct with respect to an ordering $>$ (written $> \vdash b$) if $>_b$ (the definition order in b) respects $>$ (in other words $> \subset >_b$). We abbreviate $\gtrdot_b \vdash b$ with $\vdash b$.

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

4.2 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 χ, ordering is denoted by $\geq$, greatest lower bound operation is denoted by $\wedge$. We denote by *Positive* and *Negative* the sets of positive and negative degrees, respectively. The compatibility condition means that for all $\chi_1 \in Positive$ and $\chi_2 \in Negative$, we have $\chi_1 \geq \chi_2$.

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 $N_1 \xrightarrow{\chi_2} N_3$ for each pair of edges $N_1 \xrightarrow{\chi_1} N_2$ and $N_2 \xrightarrow{\chi_2} N_3$ 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 δ. 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 $N_1 \xrightarrow{\chi}{}^{+} N_2$ for a non-empty path of degree χ. Also, the concatenation of a non-empty path $N_1 \xrightarrow{\chi_1}{}^{+} N_2$ and a possibly empty path δ from N_2 to N_3 is written $N_1 \xrightarrow{\chi_1}{}^{+} N_2; N_2 \xrightarrow{\chi_2}{}^{*}$, where χ_2 is χ_1 if δ is empty, and the degree of δ 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 $N_1 \xrightarrow{\chi_1}{}^{+} N_2 \xrightarrow{\chi_2}{}^{*}$.

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 $\trianglerighteq$. Their strict versions are denoted by $\triangleright$. For any dependency graph $\rightarrow$, let $\overset{Negative}{\rightarrow}$ 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 $\trianglerighteq$, or respects the order $\trianglerighteq$, if $\overset{Negative}{\rightarrow^+}$ is compatible with $\triangleright$. We write it $\vdash (\rightarrow, \trianglerighteq)$ (or $\vdash (\rightarrow, \triangleright)$).*

We have the following equivalence.

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

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 $\trianglerighteq$ on nodes, a edge $N_1 \xrightarrow{\chi} N_2$, or a path $N_1 \xrightarrow{\chi}{}^{*}$ is said backward if $N_2 \trianglerighteq N_1$.*

Proof

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

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

We require the degree function to meet the following condition.

Hypothesis 2 (Degree function)

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

4.3.1 Modeling the reduction with graphs

Definition 13 (Mixin redex) *Mixin redexes* $e_{\uparrow}$ *are defined by*

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

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)

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

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

Proof By case analysis on the reduction.

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

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

Other cases similar.

□

4.3.2 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 δ for paths in graphs. The minimum degree of a path $\delta = X \xrightarrow{\chi_1} N_1 \ldots N_{n-1} \xrightarrow{\chi_n} Y$ is $\chi = \bigwedge_{1 \leq i \leq n} \chi_i$.

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

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

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

Induction. By induction hypothesis.

□

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

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

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

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

Proof

Sum. It is obvious that $\lfloor \rightarrow_1 \rfloor \cup \lfloor \rightarrow_2 \rfloor \subset \lfloor \rightarrow_1 \cup \rightarrow_2 \rfloor$, since $\rightarrow_1 \subset \rightarrow_1 \cup \rightarrow_2$ and lift is monotone.

Now, an edge between names X and Y in $\lfloor \rightarrow_1 \cup \rightarrow_2 \rfloor$, implies the existence of a path between X and Y through variables only, in $\rightarrow_1 \cup \rightarrow_2$, but as variables cannot interact, this path is entirely in either one of the two subgraphs.

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

$$\begin{array}{rcl} \to_1 & = & \to !X \rhd x \\ \to'_1 & = & \lfloor \to_1 \rfloor \end{array} \qquad \text{and} \qquad \begin{array}{rcl} \to_2 & = & \lfloor \to \rfloor \\ \to'_2 & = & \to_2 !X \rhd x \\ \to''_2 & = & \lfloor \to'_2 \rfloor \end{array}$$

First, notice that both in $\to'_1$ and $\to''_2$, no edge starts from X, and edges arriving to X come from paths to X through x with degree $\frownie$ in $\to_1$ and $\to'_2$, respectively, so they have degree $\frownie$.

- $\to''_2 \subset \to'_1$.
  - Let $Y \xrightarrow{\chi}{}''_2 X$, with $X \neq Y$. Necessarily, $\chi = \frownie$.
    This implies that there exists a path of $\to'_2$ of the shape

    $$Y \xrightarrow{\chi_0}{}'_2 x \xrightarrow{\chi_1}{}'_2 x \dots \xrightarrow{\chi_n}{}'_2 x \xrightarrow{\frownie}{}'_2 X,$$

    because x is the only variable in $\to'_2$.
    n could be zero, in which case we would have $Y \xrightarrow{\frownie}{}'_2 X$.
    But this means that we have $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \dots \xrightarrow{\chi_n}{}_2 X$.
    So by definition of $\lfloor \rfloor$, we have $Y \xrightarrow{\chi_0}{}^{\Box} X \xrightarrow{\chi_1}{}^{\Box} X \dots \xrightarrow{\chi_n}{}^{\Box} X$.
    So, we have $Y \xrightarrow{\chi_0}{}^{\Box}_1 x \xrightarrow{\chi_1}{}^{\Box}_1 x \dots \xrightarrow{\chi_n}{}^{\Box}_1 x \xrightarrow{\frownie}{}_1 X$,
    and therefore $Y \xrightarrow{\frownie}{}'_1 X$.
  - Let $Y \xrightarrow{\chi}{}''_2 Z$, with Y and Z different from X. Then

    $$Y \xrightarrow{\chi_0}{}'_2 x \xrightarrow{\chi_1}{}'_2 x \dots x \xrightarrow{\chi_n}{}'_2 Z,$$

    because x is the only variable in $\to'_2$.
    We have $\chi = \bigwedge_{0 \leq i \leq n} \chi_i$. n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}'_2 Z$.
    We can deduce $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \dots X \xrightarrow{\chi_n}{}_2 Z$,
    so $Y \xrightarrow{\chi_0}{}^{\Box} X \xrightarrow{\chi_1}{}^{\Box} X \dots X \xrightarrow{\chi_n}{}^{\Box} Z$,
    and therefore $Y \xrightarrow{\chi_0}{}^{\Box}_1 x \xrightarrow{\chi_1}{}^{\Box}_1 x \dots x \xrightarrow{\chi_n}{}^{\Box}_1 Z$.
    So we have $Y \xrightarrow{\chi}{}'_1 Z$.
- $\to'_1 \subset \to''_2$.
  - Let $Y \xrightarrow{\chi}{}'_1 X$, with $Y \neq X$. We have

    $$Y \xrightarrow{\chi_0}{}^{\Box}_1 x \xrightarrow{\chi_1}{}^{\Box}_1 x \dots x \xrightarrow{\chi_n}{}^{\Box}_1 x \xrightarrow{\frownie}{}_1 X,$$

    where for all i, $\xrightarrow{\chi_i}{}^{\Box}_1$ does not go through x.
    As above, we have $\chi = \frownie$ and n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}^{\Box}_1 x \xrightarrow{\frownie}{}_1 X$.
    This implies that $Y \xrightarrow{\chi_0}{}^{\Box} X \xrightarrow{\chi_1}{}^{\Box} X \dots X \xrightarrow{\chi_n}{}^{\Box} X$.
    Therefore, $Y \xrightarrow{\chi_0}{}_2 X \xrightarrow{\chi_1}{}_2 X \dots X \xrightarrow{\chi_n}{}_2 X$.
    So, $Y \xrightarrow{\chi_0}{}'_2 x \xrightarrow{\chi_1}{}'_2 x \dots x \xrightarrow{\chi_n}{}'_2 x \xrightarrow{\frownie}{}'_2 X$,
    and so $Y \xrightarrow{\frownie}{}''_2 X$.
  - Let $Y \xrightarrow{\chi}{}'_1 Z$, with Y and Z different from X.
    We deduce $Y \xrightarrow{\chi_0}{}^{\Box}_1 x \xrightarrow{\chi_1}{}^{\Box}_1 x \dots x \xrightarrow{\chi_n}{}^{\Box}_1 Z$,
    where for all i, $\xrightarrow{\chi_i}{}^{\Box}_1$ does not go through x.
    As above, we have $\chi = \bigwedge_{0 \leq i \leq n} \chi_i$ and n could possibly be 0, in which case the path would rather look like $Y \xrightarrow{\chi_0}{}^{\Box}_1 Z$.

Then, $Y \xrightarrow{\chi_0}^{\square} X \xrightarrow{\chi_1}^{\square} X \ldots X \xrightarrow{\chi_n}^{\square} Z$,
and so $Y \xrightarrow{\chi_0}_2 X \xrightarrow{\chi_1}_2 X \ldots X \xrightarrow{\chi_n}_2 Z$,
which leads to $Y \xrightarrow{\chi_0}_2' x \xrightarrow{\chi_1}_2' x \ldots x \xrightarrow{\chi_n}_2' Z$,
and so $Y \xrightarrow{\chi}_2'' Z$.

Other cases. Easy.

□

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

Proof

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

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

□

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

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

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

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

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

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

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

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

4.3.3 Manipulation of recursive bindings

Definition 15 (Graph comparison) *We define $\to_1 < \to_2$ by*

- *for all $N_1 \overset{\frownie}{\longrightarrow}_2 N_2$, there exists $N_1 \overset{\frownie}{\longrightarrow}{}_1^+ N_2$*
- *for all $N_1 \overset{\smiley}{\longrightarrow}_2 N_2$, there exists $N_1 \xrightarrow{\chi}_1 N_2$.*

In particular, if $\to_2 \subset \to_1$, then $\to_1 < \to_2$; and if for all edge in $\to_2$ there exists an edge with the same ends and an inferior degree in $\to_1$, then $\to_1 < \to_2$. 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 ($\mathrm{dom}(b_1) = \mathrm{dom}(b_2)$), they define variables in the same order ($>_b = >_{b'}$), the dependencies and shapes of b_1 are more restrictive than those of b_2 ($\to_{b_1} < \to_{b_2}$, and for all $x \in \mathrm{dom}(b_2)$, if $b_2(x) \notin$ Predictable, then $b_1(x) \notin$ 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 $\gtrdot_{b'} \vdash b$, then $\vdash b$.*

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

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

□

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

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

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

Definition 17 (Paths) *For a path $\delta = (N_0 \xrightarrow{\chi_1} \dots \xrightarrow{\chi_n} N_n)$, we define the degree of δ as χ_n, and we write δ^χ for a path of degree χ, and $\delta \subset \to$ if δ is a path of $\to$.*

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 $\delta_1^{\chi_1}, (x, y, \chi), \delta_2^{\chi_2}$.

Proposition 5 (Let rec internal dependencies)

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

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

Proof

Let $X \neq Y$, $b = (x_1 = e_1 \ldots x_n = e_n)$, and $o = (Output(b), Y \triangleright \_ = e)$.

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

□

Proposition 6 (Merging nested bindings)

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

Let δ a path of $\rightarrow_{b'}$, from x_1 to x_2, of degree χ.

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

Proof By induction on the length of δ.

Base δ is an edge.

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

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

1. $x_1, x_2 \in dom(b)$.

- If δ only has nodes in $dom(b)$, let (x_3, x_2, χ) be its last edge. By induction hypothesis there is a path $\delta_1^{\chi'}$ from x_3 to x_2 with $\chi' \leq \chi$ in $\rightarrow_b$, and a path $\delta_2^{\chi''}$ from x_1 to x_3, so $\delta_1^{\chi''}, \delta_2^{\chi'} \subset \rightarrow_b$, with degree $\chi' \leq \chi$.
- Otherwise, let x_5 be the last node of δ in $dom(b_2)$. The next node is necessarily x. Let x_3 be the last node of δ in $dom(b)$ before x_5. Let x_4 be the next node. (It is in $dom(b_2)$.) We have

  $$\delta = \delta_1^{\chi_1}, (x_3, x_4, \chi_4), \delta_2^{\chi_2}, (x_5, x, \chi_5), \delta_3^{\chi_3},$$

  with $\delta_2^{\chi_2} \subset \rightarrow_{b_2}$. Let now $o = (Output(b_2), Y \triangleright \_ = e)$ and consider the structure $\langle X \triangleright x_3; o\rangle$. Its concrete dependency graph is $\rightarrow_{\langle X \triangleright x_3; o\rangle}$ and contains a path $(X, x_4, \chi_4), \delta_2^{\chi_2}, (x_5, Y, \chi_5)$.
  So by hypothesis 2, we have $\chi_5' = Degree(x_3, \mathsf{let\ rec}\ \ b_2\ \ \mathsf{in}\ \ e) \leq \chi_5$, so there is and edge $x_3 \xrightarrow{\chi_5'}_b x$.
  Then, applying the induction hypothesis to δ_1 and δ_3 if not empty, we obtain two paths $\delta_1'^{\chi_1'}$ and $\delta_3'^{\chi_3'}$ of $\rightarrow_b$, and so $\delta_1'^{\chi_1'}, (x_3, x, \chi_5'), \delta_3'^{\chi_3'}$ is a path of $\rightarrow_b$, with a degree $\chi_5' \leq \chi_5$ if δ_3 is empty, and a degree $\chi_3' \leq \chi_3$ otherwise.

2. $x_1 \in dom(b), x_2 \in dom(b_2)$. Let x_3 be the last node of δ in $dom(b)$, and x_4 the next one. δ is of the shape $\delta_1^{\chi_1}, (x_3, x_4, \chi_3), \delta_2^{\chi_2}$, with the nodes of δ_2 in $dom(b_2)$. δ_1 and δ_2 could be empty. As above, by lemma 5, we have $\chi_3' = Degree(x_3, \mathsf{let\ rec}\ \ b_2\ \ \mathsf{in}\ \ e) \leq Degree(x_3, b_2(x_4)) = \chi_3$, so we have an edge $x_3 \xrightarrow{\chi_3'}_b x$.
   - If δ_2 is empty, $n > 1$, so δ_1 is non-empty, and applying induction hypothesis to δ_1, we obtain $\delta_1'^{\chi_1'}$ with same ends, and therefore obtain a path in $\rightarrow_b$ with same ends as δ, and with degree $\chi_3' \leq \chi_3 = \chi$.
   - Otherwise, $\delta_2{}^{\chi_2} \subset \rightarrow_{b_2}$. Let $X \neq Y$, $\iota = X \triangleright x_3$, and $o = Output(b_2), Y \triangleright \_ = e$. We obtain a path $(x_3, x_4, \chi_3), \delta_2{}^{\chi_2}$ in $\rightarrow_{\langle \iota; o\rangle}$ with same ends as δ, and with degree $\chi_2 = \chi$. So, if δ_1 is empty, we have in both cases a path from x_3 to x in b, with degree $\chi_2' \leq \chi$. Otherwise, by induction hypothesis, we obtain $\delta_1'^{\chi_1'}$ with $\chi_1' \leq \chi_1$, and reason exactly as above.

3. $x_1 \in dom(b_2), x_2 \in dom(b)$. Assume $\chi = \frownie$. The first node of δ not in $dom(b_2)$ is necessarily x. Let x_3 be the node just before it. δ has the shape $\delta_1^{\chi_1}, (x_3, x, \chi_3), \delta_2^{\chi_2}$. If δ_2 is empty, we have $x_2 = x$ which is clearly in $\{x\} \cup dom(b_3)$. Otherwise, apply induction hypothesis to obtain a path $\delta_2'^{\chi_2'}$ with the same ends as δ_2 and $\chi_2' \leq \chi_2$. But here $\chi = \chi_2 = \frownie$ so $\chi_2' = \frownie$. As $G_b \vdash b$, x_2 must be defined after x in b, so it must be in $dom(b_3)$.

4. $x_1, x_2 \in dom(b_2)$. If all the nodes are in $dom(b_2)$, then $\delta \subset \rightarrow_{b_2}$ directly. Otherwise, the first node not in $dom(b_2)$ in δ is necessarily x. Let x_3 be the node just before it. δ has to continue after x, because it has to go back to a node in $dom(b_2)$, by hypothesis. Let x_4 be the node just after the first occurence of x. δ has the shape $\delta_1^{\chi_1}, (x_3, x, \chi_3), (x, x_4, \chi_4), \delta_2^{\chi_2}$.
   - If δ_2 is empty, then as $Degree(x, b_2(x_4)) = \chi_4$, by lemma 5 there exists an edge $x \xrightarrow{\chi_4'}_b x$, with $\chi_4' \leq \chi_4$. But here $\chi_4 = \chi$, so we are in the second case and $x \xrightarrow{\chi_4'}{}^{+}_{b} x$ with $\chi_4' \leq \chi$.
   - Otherwise, by induction hypothesis on $(x, x_4, \chi_4), \delta_2^{\chi_2}$, we obtain a path $\delta_2'^{\chi_2'} \subset \rightarrow_b$, from x to x and $\chi_2' \leq \chi_2$, which means that $x \xrightarrow{\chi_2'}{}^{+}_{b} x$, and that is enough.

□

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

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

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

□

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

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

Proof Let $\delta^{\text{☹}}$ 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 $dom(b_2) \perp FV(b_1)$.

□

4.4 Soundness

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

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

Proof By induction on the typing derivation.

Struct. $e = \langle \iota; o \rangle$ and $M = \langle I; O; G \rangle$. By syntactic correctness, $dom(\iota) \perp Names(o)$, so $dom(I) \perp dom(O)$. Moreover, the targets of G, by construction of $\rightarrow_{\langle \iota; o \rangle}$, and $\rightarrow^{\square}_{\langle \iota; o \rangle}$, are in $dom(O)$, and by typing $\vdash G_{\langle \iota; o \rangle}$, 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 $dom(O_1 \uplus O_2)$.

Freeze. $e = e' \,!\, X$, $\Gamma \vdash e' : \langle I; O; G \rangle$, and $M = \langle I; O; G \,!\, X \rangle$. The only difficulty is to show that the targets of $G \,!\, X$ are in $\mathit{dom}(O)$, 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 $\mathit{Targets}(G_{|\mathcal{N}}) \subset \mathit{dom}(O_{|\,\mathcal{N}})$, but by induction hypothesis $\mathit{Targets}(G) \subset \mathit{dom}(O)$, and as $\mathit{Targets}(G_{|\mathcal{N}}) = \mathit{Targets}(G) \cap \mathcal{N}$, we have $\mathit{Targets}(G_{|\mathcal{N}}) \subset \mathit{dom}(O) \cap \mathcal{N} = \mathit{dom}(O_{|\,\mathcal{N}})$.

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 $\vdash I$, $\vdash O$, $\vdash G$, $\mathit{dom}(I) \perp \mathit{dom}(O)$, and $\mathit{Targets}(G) \subset \mathit{dom}(O)$. We can deduce that $\langle I; O_{|\,X_1 \ldots X_n}; G_{:X_1 \ldots X_n} \rangle$ is well-formed, since $\mathit{Targets}(G_{:X_1 \ldots X_n}) \subset \{X_1 \ldots X_n\}$ and by typing $\{X_1 \ldots X_n\} \subset \mathit{dom}(O)$. The other conditions are easy, and hide is similar.

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

By induction hypothesis, $\mathit{dom}(I) \perp \mathit{dom}(O)$, $\mathit{Targets}(G) \subset \mathit{dom}(O)$, $\vdash I$, $\vdash O$ 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 $\mathit{dom}(I\{r\}) = (\mathit{dom}(I) \setminus \mathit{dom}(r)) \uplus r(\mathit{dom}(I))$ and
$\mathit{dom}(O\{r\}) = (\mathit{dom}(O) \setminus \mathit{dom}(r)) \uplus r(\mathit{dom}(O))$.

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

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

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

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

$$\mathit{Targets}(G\{r\}) \subset r(\mathit{dom}(O)) \cup (\mathit{dom}(O) \setminus \mathit{dom}(r)) = \mathit{dom}(O\{r\}).$$

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

Other cases. Easy.

□

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

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

Now, typing is preserved by the computational contraction rules.

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

Proof By case analysis on the contraction step.

- Sum. Assume $e = \langle \iota_1 ; o_1 \rangle + \langle \iota_2 ; o_2 \rangle$, and $\Gamma \vdash e : M$. By typing we have $\Gamma \vdash \langle \iota_1 ; o_2 \rangle : \langle I_1 ; O_1 ; G_1 \rangle$, $\Gamma \vdash \langle \iota_2 ; o_2 \rangle : \langle I_2 ; O_2 ; G_2 \rangle$, and $M = \langle I ; O ; G \rangle$, with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = O_1 \uplus O_2$, $G = G_1 \cup G_2$, and $\vdash G$. We have $e' = \langle \iota ; o \rangle$, where $\iota = (\iota_1 \cup \iota_2) \setminus \mathit{Input}(o_1, o_2)$, $o = o_1 \succ_\iota o_2$, with $\langle \iota_1 ; o_1 \rangle \Bumpeq \langle \iota_2 ; o_2 \rangle$.

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

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

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

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

    Then, $I = I'_1 \uplus I'_2$, with $I'_1 = I_1 \setminus O_2$ and $I'_2 = I_2 \setminus (I_1 \cup O_1)$ and we obtain $I \circ \iota^{-1} = (I'_1 \circ \iota^{-1}) \uplus (I'_2 \circ \iota^{-1}) = (I'_1 \circ \iota_1^{-1}) \uplus (I'_2 \circ \iota_2^{-1})$.

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

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

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

    $$\begin{array}{rcl} \Gamma' & = & \Gamma\langle \Gamma_1 \uplus ((I_1 \setminus O_2) \circ \iota_1^{-1}) \uplus (I'_2 \circ \iota_2^{-1}) \uplus (I_1 \cap O_2) \circ \iota_1^{-1} \uplus \Gamma_{2 \setminus P} \rangle \\ & = & \Gamma\langle \Gamma_1 \uplus I_1 \circ \iota_1^{-1} \uplus (I'_2 \circ \iota_2^{-1}) \uplus \Gamma_{2 \setminus P} \rangle. \end{array}$$

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

- Lift.

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

  $$\dfrac{\dfrac{\vdash \Gamma_b \qquad \vdash b \qquad \dfrac{\vdots}{\Gamma\langle \Gamma_b \rangle \vdash b : \Gamma_b} \qquad \dfrac{\vdots}{\Gamma\langle \Gamma_b \rangle \vdash e_1 : M_1}}{\Gamma \vdash \mathsf{let\ rec}\ b\ \mathsf{in}\ e_1 : M_1} \qquad \dfrac{\vdots}{\Gamma \vdash e_2 : M_2} \qquad \begin{array}{l} I_1 \uplus O_1 \Bumpeq I_2 \uplus O_2 \\ \vdash G_1 \cup G_2 \end{array}}{\Gamma \vdash (\mathsf{let\ rec}\ b\ \mathsf{in}\ e_1) + e_2 : M}$$

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

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

  $$\dfrac{\begin{array}{l} \vdash \Gamma_b \\ \vdash b \end{array} \qquad \dfrac{\vdots}{\Gamma\langle \Gamma_b \rangle \vdash b : \Gamma_b} \qquad \dfrac{\begin{array}{l} I_1 \uplus O_1 \Bumpeq I_2 \uplus O_2 \\ \vdash G_1 \cup G_2 \end{array} \qquad \dfrac{\vdots}{\Gamma\langle \Gamma_b \rangle \vdash e_1 : M_1} \qquad \dfrac{\vdots}{\Gamma\langle \Gamma_b \rangle \vdash e_2 : M_2}}{\Gamma\langle \Gamma_b \rangle \vdash e_1 + e_2 : M}}{\Gamma \vdash \mathsf{let\ rec}\ b\ \mathsf{in}\ e_1 + e_2 : M}$$

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

  By typing, we have a derivation of the shape

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

  with $O = \Gamma_o \circ \mathit{Input}(o)$ and $G = \lfloor \rightarrow_{\langle \iota; o\rangle} \rfloor$.

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

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

  For x and y, we easily derive too that

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

  So we have

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

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

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

- Delete. Let $e = \langle \iota; o\rangle_{|-X_1 \ldots X_n}$, with $\mathcal{N} = \{X_1 \ldots X_n\}$, we have

$$\dfrac{\dfrac{dom(\iota) = dom(I) \qquad \vdash I \qquad \vdash \Gamma_o \qquad \vdash \rightarrow_{\langle \iota; o\rangle} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle \iota; o\rangle : \langle I; O; G\rangle}}{\Gamma \vdash e : M}$$

  with $M = \langle I'; O'; G'\rangle = \langle I \uplus O_{|\,\mathcal{N}}; O_{\setminus\mathcal{N}}; G_{|-\mathcal{N}}\rangle$ and $G = \lfloor \rightarrow_{\langle \iota; o\rangle} \rfloor$. But necessarily, we have $e' = \langle \iota'; o'\rangle = \langle \iota, \mathit{Input}(o)_{|\,\mathcal{N}}; o_{\setminus\mathcal{N}}\rangle$.

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

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

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

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

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

  By typing, we have $\Gamma \vdash \langle \iota; o\rangle : \langle I; O; G\rangle$, and $M = \langle I'; O'; G'\rangle$, with $I' = I \cup O_{\setminus\mathcal{N}}$, $O' = O_{|\,\mathcal{N}}$, and $G' = G_{|\mathcal{N}}$.

  But we can derive $\Gamma \vdash e'' : \langle I''; O''; G''\rangle$, with $I'' = I \cup O_{|\,\mathcal{N}'} = I \cup O_{\setminus\mathcal{N}} = I'$, $O'' = O_{\setminus\mathcal{N}'} = O_{|\,\mathcal{N}} = O'$, and $G'' = G_{|-\mathcal{N}'} = G_{|\mathcal{N}} = G'$, so we derive $\Gamma \vdash e'' : M$, and may apply the same process as above to deduce $\Gamma \vdash e' : M$.

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

$$\dfrac{\dfrac{\vdash \to_{\langle\iota;o\rangle} \qquad \begin{array}{c}\vdash I \qquad \vdash \Gamma_o\\ dom(I) = dom(\iota)\end{array} \qquad \Gamma\langle I \circ \iota^{-1} \uplus \Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle\iota;o\rangle : \langle I;O;G\rangle} \\ (cod(r) \setminus dom(r)) \perp (dom(I) \cup dom(O))}{\Gamma \vdash e : M}$$

with $M = \langle I';O';G'\rangle = \langle I\{r\};O\{r\};G\{r\}\rangle$, $G = \lfloor \to_{\langle\iota;o\rangle} \rfloor$, and $O = \Gamma_o \circ Input(o)$.

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

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

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

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

$$\dfrac{\vdash \to_{\langle\iota';o'\rangle} \qquad \begin{array}{c}\vdash I' \qquad \vdash \Gamma_o\\ dom(\iota') = dom(I')\end{array} \qquad \Gamma\langle I' \circ {\iota'}^{-1} \uplus \Gamma_o\rangle \vdash o' : \Gamma_o}{\Gamma \vdash \langle\iota';o'\rangle : \langle I';O';G'\rangle}$$

- Close. Let $e = \mathsf{close}\langle\epsilon;o\rangle$. We have $e' = \mathsf{let\ rec}\ \ Bind(\overline{o})\ \ \mathsf{in}\ \ Record(o)$, and $\vdash Bind(\overline{o})$, and by typing

$$\dfrac{\dfrac{\vdash \Gamma_o \qquad \to_{\langle\epsilon;o\rangle} \qquad \Gamma\langle\Gamma_o\rangle \vdash o : \Gamma_o}{\Gamma \vdash \langle\epsilon;o\rangle : \langle\emptyset;O;G\rangle}}{\Gamma \vdash e : M}$$

with $M = \{O\}$, $G = \lfloor \to_{\langle\epsilon;o\rangle} \rfloor$, and $O = \Gamma_o \circ Input(o)$.

Let

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

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

We have

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

so it is ok.

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

  $$\frac{\dfrac{\vdash \rightarrow_{\langle \iota ; o \rangle} \qquad \begin{array}{c} \vdash I \qquad \vdash \Gamma_o \\ \mathit{dom}(I) = \mathit{dom}(\iota) \end{array} \qquad \Gamma \langle I \circ \iota^{-1} \uplus \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash e_0 : \langle I ; O ; G \rangle} \qquad \mathcal{N} \subset \mathit{dom}(O)}{\Gamma \vdash e : \langle I ; O_{|\mathcal{N}} ; G' \rangle}$$

  with $G = \lfloor \rightarrow_{\langle \iota ; o \rangle} \rfloor$, $G' = \lfloor G_{:\mathcal{N}} \rfloor$,
  and $O = \Gamma_o \circ \mathit{Input}(o)$.

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

  The typing of o' is exactly as the one for o, so we obtain that e' has type $\langle I ; O' ; G' \rangle$, with $O' = \Gamma_o \circ \mathit{Input}(o')$. But $\mathit{Input}(o') = \mathit{Input}(o)_{|\mathcal{N}}$, 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 $e_0 = \langle \iota ; o \rangle$ and $e = e_{0X \succ Y}$, with $o = (o_1, X[z^*] \triangleright x = e_1, o_2)$. We have $e' = \langle \iota' ; o' \rangle = \langle \iota, X \triangleright x ; o_1, X[z^*] \triangleright y = e_1, o_2 \rangle$ for a fesh y.

  The typing derivation is of the shape

  $$\frac{\dfrac{\vdash \rightarrow_{\langle \iota ; o \rangle} \qquad \begin{array}{c} \vdash I \qquad \vdash \Gamma_o \\ \mathit{dom}(I) = \mathit{dom}(\iota) \end{array} \qquad \Gamma \langle I \circ \iota^{-1} \uplus \Gamma_o \rangle \vdash o : \Gamma_o}{\Gamma \vdash e_0 : \langle I ; O ; G \rangle} \qquad Y \notin \mathit{dom}(O) \cup \mathit{dom}(I)}{\Gamma \vdash e_{X \succ Y} : \langle I \uplus \{X : O(X)\} ; O\{X \mapsto Y\} ; G_{X \succ Y} \rangle}$$

  with $G = \lfloor \rightarrow_{\langle \iota ; o \rangle} \rfloor$, and $O = \Gamma_o \circ \mathit{Input}(o)$.

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

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

□

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

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

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

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

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

  $$\frac{\dfrac{\vdots}{\forall (Y = f) \in (s_v, s), \Gamma \vdash f : O(Y)} \qquad \dfrac{\vdots}{\Gamma \vdash e : M'}}{\Gamma \vdash \{\mathbb{S}[e]\} : M}$$

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

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

  $$\frac{\dfrac{\vdots}{\forall (Y = f) \in (s_v, s), \Gamma \vdash f : O(Y)} \qquad \dfrac{\vdots}{\Gamma \vdash e' : M'}}{\Gamma \vdash \{\mathbb{S}[e]\} : M}$$

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

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

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

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

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

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

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

□

This property is true for multiple lift contexts as well.

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

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

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

Proof Trivial. □

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

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

Then $\Gamma \vdash \mathbb{E}[e'] : M$*.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

Lemma 11 (Evaluation context) *If $e \rightsquigarrow_c e'$, and $\Gamma \vdash \mathbb{E}[e] : M$, then $\Gamma \vdash \mathbb{E}[e'] : M$.*

Proof By lemma 10. □

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

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

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

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

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

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

For this we remark that

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

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

- For variables $z \in FV(v) \setminus \mathit{Capt}_\Box(\mathbb{F}[\mathbb{N}\,])$, we have $\chi = \frownie$.

- For variables $z \notin FV(v) \setminus Capt_{\Box}(\mathbb{F}[\mathbb{N}])$, we have $\chi = Degree(z, \mathbb{F}[\mathbb{N}])$.

Therefore, we have

$$\begin{aligned} \rightarrow_{b'} \subset \; & \rightarrow_b \\ & \cup \{ z \xrightarrow{\frownie} y \mid z \in FV(v) \setminus Capt_{\Box}(\mathbb{F}[\mathbb{N}]) \} \\ & \cup \{ z \xrightarrow{\chi} y \mid z \in FV(\mathbb{F}[\mathbb{N}[v]]) \cap (FV(v) \setminus Capt_{\Box}(\mathbb{F}[\mathbb{N}])), \chi = Degree(z, \mathbb{F}[\mathbb{N}]) \} \end{aligned}$$

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

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

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

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

□

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

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

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

- $\mathbb{E} = \mathbb{F}$, impossible.
- $\mathbb{E} = \mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{F}$. By corollary 5.
- $\mathbb{E} = \mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ e$. Let $b = \mathbb{B}[\mathbb{F}[\mathbb{N}[x]]]$, $b' = \mathbb{B}[\mathbb{F}[\mathbb{N}[v]]]$, and $\mathbb{B} = b_v, y = \Box, b_1$.
  The derivation has the shape

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

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

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

$$\frac{\vdash b' \qquad \dfrac{\vdots}{\forall y \in dom(b'), \Gamma\langle\Gamma_b\rangle \vdash b' : \Gamma_b(y)}}{\Gamma \vdash \mathbb{E}[\mathbb{N}[v]] : M}$$

□

Type preservation along the IM rule is proven.

Lemma 14 (Internal merge)

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

Proof

We have a derivation of the shape

$$\dfrac{\dfrac{\forall y \neq x \;\; \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash b(y) : \Gamma_b(y)} \qquad \dfrac{\begin{array}{c}\vdash \Gamma_{b_1}\\ \vdash b_1\end{array} \;\; \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash b_1 : \Gamma_{b_1}} \;\; \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle\langle\Gamma_{b_1}\rangle \vdash e_1 : \Gamma_b(x)}}{\Gamma\langle\Gamma_b\rangle \vdash \mathsf{let\ rec}\;\; b_1 \;\;\mathsf{in}\;\; e_1 : \Gamma_b(x)}}{\Gamma\langle\Gamma_b\rangle \vdash b : \Gamma_b} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_b\rangle \vdash f : M} \quad \begin{array}{l}\vdash \Gamma_b\\ \vdash b\end{array}}{\Gamma \vdash \mathsf{let\ rec}\;\; b_v, x = (\mathsf{let\ rec}\;\; b_1 \;\;\mathsf{in}\;\; e_1), b_2 \;\;\mathsf{in}\;\; f : M}$$

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

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

Moreover, by weakening, we have

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

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

$$\dfrac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle : f : M}$$

and we have

$$\dfrac{\forall y \in dom(b') \;\; \dfrac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle \vdash b'(y) : \Gamma_{b'}(y)} \qquad \dfrac{\vdots}{\Gamma\langle\Gamma_{b'}\rangle : f : M} \qquad \begin{array}{l}\vdash \Gamma_{b'}\\ \vdash b'\end{array}}{\Gamma \vdash \mathsf{let\ rec}\;\; b' \;\;\mathsf{in}\;\; f : M}$$

□

Next, rule EM is examined.

Lemma 15 (External merge)
If $dom(b) \perp (dom(b_v) \cup FV(b_v))$,
$e_0 = \mathsf{let\ rec}\;\; b_v \;\;\mathsf{in}\;\; \mathsf{let\ rec}\;\; b \;\;\mathsf{in}\;\; e$,
$e'_0 = \mathsf{let\ rec}\;\; b_v, b \;\;\mathsf{in}\;\; e, \mathit{and}$
$\Gamma \vdash e_0 : M$,
then $\Gamma \vdash e'_0 : M$.

Proof The typing derivation for e_0 has the shape

$$\dfrac{\vdash b_v \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle \vdash b_v : \Gamma_1} \qquad \dfrac{\vdash b \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle\langle\Gamma_2\rangle \vdash b : \Gamma_2} \quad \dfrac{\vdots}{\Gamma\langle\Gamma_1\rangle\langle\Gamma_2\rangle \vdash e : \Gamma_2}}{\Gamma\langle\Gamma_1\rangle \vdash \mathsf{let\ rec}\;\; b \;\;\mathsf{in}\;\; e : \Gamma_1}}{\Gamma \vdash e_0 : M}$$

By lemma 5, we have $\vdash b_v, b$.

So by weakening we can reconstruct the derivation.

□

We can now state the subject reduction property.

Lemma 16 (Subject reduction) *If $e \longrightarrow e'$ and $\Gamma \vdash e : M$, 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 $e = \mathbb{E}[\mathbb{N}[x]]$ with $x \notin Capt_\Box(\mathbb{E}[\mathbb{N}])$, or there exists e' such that $e \longrightarrow e'$.*

Proof By induction on e.

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

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

   Otherwise, by induction hypothesis, we are in one of the two following cases.
   - $f \longrightarrow f'$. By case analysis on the reduction:
     - EM. Then rule EM applies for e as well.
     - Subst or Context.
       We have $f = \mathbb{E}[g]$ and $f' = \mathbb{E}[g']$. If $\mathbb{E} = \mathsf{let\ rec}\ {b_v}'\ \mathsf{in}\ \mathbb{F}'$ or $\mathbb{E} = \mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ g$, then rule EM applies, and otherwise, the same rule applies for e since $\mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{E}$ is an evaluation context.
   - $f = \mathbb{E}[\mathbb{N}[x]]$, with $x \notin Capt_\Box(\mathbb{E}[\mathbb{N}])$.
     If $\mathbb{E} = \mathsf{let\ rec}\ {b_v}'\ \mathsf{in}\ \mathbb{F}$ or $\mathbb{E} = \mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ g$, then rule EM applies, and otherwise, $\mathbb{E}$ is of the shape $\mathbb{F}$ and $f = \mathbb{F}[\mathbb{N}[x]]$, $e = \mathsf{let\ rec}\ b_v\ \mathsf{in}\ \mathbb{F}[\mathbb{N}[x]]$. If $x \in dom(b_v)$, then rule Subst applies, and otherwise $e = \mathbb{E}_0[\mathbb{N}[x]]$ with $x \notin Capt_\Box(\mathbb{E}_0)$.

   (b) Otherwise, b is not evaluated, so b is of the shape $b_v, y = g, b_1$, where g is not a value.
   - If g is a result, then it is of the shape $\mathsf{let\ rec}\ {b_v}'\ \mathsf{in}\ v$, and by internal merge,
     $e \longrightarrow \mathsf{let\ rec}\ b_v, {b_v}', y = v, b_1\ \mathsf{in}\ f$.
   - Otherwise, by induction hypothesis:
     - If $g \longrightarrow g'$, by case on the reduction.
       - EM: then rule IM applies for e.
       - Context or Subst: then $g = \mathbb{E}[g_0]$ and $g' = \mathbb{E}[g'_0]$. If $\mathbb{E}$ is of the shape $\mathsf{let\ rec}\ {b_v}'\ \mathsf{in}\ \mathbb{F}$ or $\mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ g''$, then rule IM applies for e, and otherwise, the global context is an evaluation context and the same rule Context or Subst applies for e.

- If $g = \mathbb{E}[\mathbb{N}[x]]$ with $x \notin \mathit{Capt}_{\square}(\mathbb{E}[\mathbb{N}])$. By case on $\mathbb{E}$. First notice that we know that $x \notin \mathit{dom}(y = g, b_1)$, since by typing $\vdash b$ and therefore if $x \overset{\frownie}{\longrightarrow}{}^{+}_{b}\; y$, then x is defined before y in b, and $g = \mathbb{E}[\mathbb{N}[x]]$ implies the existence of an edge $x \overset{\frownie}{\longrightarrow}_{b} y$ by hypothesis 2.
  - * If $\mathbb{E} = \mathsf{let\ rec}\ b_v{}'\ \mathsf{in}\ \mathbb{F}$ or $\mathsf{let\ rec}\ \mathbb{B}[\mathbb{F}]\ \mathsf{in}\ g''$, then rule IM applies.
  - * Else, if $x \in \mathit{dom}(b_v)$, then rule SUBST applies, since the global context is an evaluation context.
  - * Else, if $x \notin \mathit{dom}(b_v, y = g, b_1)$, then e is of the shape $\mathbb{E}_0[\mathbb{N}[x]]$ with $x \notin \mathit{Capt}_{\square}(\mathbb{E}_0)$.

4. $e = e_1 + e_2$.

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

$$\dfrac{\dfrac{\begin{array}{c}\vdash \rightarrow_{\langle\iota_1;o_1\rangle}\\ \vdash I_1 \qquad \vdash \Gamma_1\\ \mathit{dom}(\iota_1) = \mathit{dom}(I_1)\end{array} \quad \dfrac{\vdots}{\Gamma\langle I_1 \circ \iota_1^{-1} \uplus \Gamma_1\rangle \vdash o_1 : \Gamma_1}}{\Gamma \vdash e_1 : \langle I_1; O_1; G_1\rangle} \qquad \dfrac{\begin{array}{c}\vdash \rightarrow_{\langle\iota_2;o_2\rangle}\\ \vdash I_2 \qquad \vdash \Gamma_2\\ \mathit{dom}(\iota_2) = \mathit{dom}(I_2)\end{array} \quad \dfrac{\vdots}{\Gamma\langle I_2 \circ \iota_2^{-1} \uplus \Gamma_2\rangle \vdash o_2 : \Gamma_2}}{\Gamma \vdash e_2 : \langle I_2; O_2; G_2\rangle}}{\Gamma \vdash e : M}$$

   with $\vdash G_1 \cup G_2$ and $I_1 \uplus O_1 \asymp I_2 \uplus O_2$ as side-conditions and

$$\begin{array}{l} G_1 = \rightarrow_{\langle\iota_1;o_1\rangle}\\ G_2 = \rightarrow_{\langle\iota_2;o_2\rangle}\\ O_1 = \Gamma_1 \circ \mathit{Input}(o_1)\\ O_2 = \Gamma_2 \circ \mathit{Input}(o_2)\end{array} \qquad \text{and} \qquad \begin{array}{l} M = \langle I; O; G\rangle\\ I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)\\ O = O_1 \uplus O_2\\ G = G_1 \cup G_2.\end{array}$$

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

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

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

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

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

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

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

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

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 *MM* 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 $S \in \mathit{Names}$ are distinguished from variables $s \in \mathit{Vars}$. Expression variables $x \in \mathit{MVars} \subset \mathit{Vars}$ are distinguished from type names $t \in \mathit{TVars} \subset \mathit{Vars}$. Expression names $X \in \mathit{MNames} \subset \mathit{Names}$ are distinguished from type names $T \in \mathit{TNames} \subset \mathit{Names}$. 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* $o = (d_1 \dots d_n)$, 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 $\{o_v\}$ is a list o_v of pairs $S \triangleright s$ of a name and a variable. For homogeneity, we consider these pairs as

Lexical conventions:

	Expression	Type	Both
Variable	x	t	s
Name	X	T	S
Labels	$L \in \mathit{Names} \uplus \{\_\}$		

Path:	p	$::= x \mid p.X$	
Expression:	e	$::= p$	Path
		$\mid \{o_v\}$	Module
		$\mid \mathsf{let\ rec}\ o\ \mathsf{in}\ e$	let rec
		$\mid \langle I; o\rangle$	Structure
		$\mid e_1 + e_2 \mid \mathsf{close}\, e$	Composition, closure
		$\mid (e : M)$	Type constraint
	o	$::= d_1 \dots d_n$	Output
	d	$::= L \triangleright x[y^*] = e \mid T \triangleright t = M$	Definition
Type:	M	$::= \star$	New type
		$\mid t \mid p.T$	Type path
		$\mid \{O\}$	Module type
		$\mid \langle I; O; \rightarrow; \rightarrowtriangle\rangle$	Mixin module type
	I, O	$::= D_1 \dots D_n$	Signature
	D	$::= L \triangleright s : M$	Declaration
	$\rightarrow$	$\subset_{\text{Fin}} \{X \xrightarrow{\chi} Y \mid X, Y \in \mathit{Names}, \chi \in \mathit{Degrees}\}$	Dynamic graph
	$\rightarrowtriangle$	$\subset_{\text{Fin}} \mathit{Names} \times \mathit{Names}$	Static graph
	$\mathit{Degrees}$	$= \{☺, ☹\}$	

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* $p = x.X_1.\dots.X_n$. 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 $I = (D_1 \dots D_n)$ 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 $T \triangleright t : \star$. 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 $p.T$, module types $\{O\}$, and mixin module types $\langle I; O; \rightarrow; \rightarrowtriangle \rangle$. 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 $\chi \in \{☺, ☹\}$. 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 $\rightarrowtriangle$ 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, $\mathsf{let\ rec}$ 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 $\mathsf{let\ rec}$ so we do not write them. Any expression is allowed as a $\mathsf{let\ rec}$ 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

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

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 $\mathit{DN}(f)$ the set of names defined by f, $\{S \mid \exists s, (S, s) \in \mathit{dom}(f)\}$, and $\mathit{DV}(f)$ the set of variables defined by f, $\{s \mid \exists s, (S, s) \in \mathit{dom}(f)\}$.

Structural equivalence We consider the expressions equivalent up to alpha-conversion of binding variables in structures, signatures and $\mathsf{let\ rec}$ 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* Γ, 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 $+$ denotes the union of finite maps, without overriding. Therefore, $\Gamma + \Gamma'$ implies $\mathit{DV}(\Gamma) \perp \mathit{DV}(\Gamma')$. A signature can be seen as an environment by forgetting labels. This will be done implicitely 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 $\rightarrowtriangle_{(O,\to)}$ of a set of declarations O with dynamic dependency graph $\to$ is defined in figure 5.2. The arrow $\to$ denotes any *concrete dynamic graph*: it is a graph over *nodes*, which are elements of $\mathit{Vars} \cup \mathit{Names}$, labeled by degrees. Further, O is any set of declarations, not necessarily defining distinct names or variables. The definition of $\rightarrowtriangle$ uses the function Nodes, which associates to a pair (L, s) of a label and a variable either s, if $L = \_$, 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 $\vdash \rightarrowtriangle_{(O,\to)}$. 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 $e =_{\text{def}} \mathsf{close}\langle; T \triangleright t = y.U, X \triangleright x = \langle\emptyset; U \triangleright u = t\rangle, Y \triangleright y = \mathsf{close}\, x\rangle$. The edges $Y \rightarrowtriangle T$ and $T \rightarrowtriangle X$ in the structure are obvious, but without the rules for prolongation with dynamic dependencies, there would not be any edge $T \rightarrowtriangle Y$. The expression reduces to

$$\begin{array}{ll}
\mathsf{let\ rec} & T \triangleright t = y.U,\\
& X \triangleright x = \langle\emptyset; U \triangleright u = t\rangle,\\
& Y \triangleright y = \begin{array}[t]{l}\mathsf{let\ rec}\, U \triangleright u = t\\ \mathsf{in}\, \{U \triangleright u' = u\}\end{array}\\
\multicolumn{2}{l}{\mathsf{in}\, \{T \triangleright t' = t, X \triangleright x' = x, Y \triangleright y' = y\}}
\end{array}
\quad \text{and then} \quad
\begin{array}{ll}
\mathsf{let\ rec} & T \triangleright t = y.U,\\
& X \triangleright x = \langle\emptyset; U \triangleright u = t\rangle,\\
& U \triangleright u = t,\\
& Y \triangleright y = \{U \triangleright u' = u\}\\
\multicolumn{2}{l}{\mathsf{in}\, \{T \triangleright t' = t, X \triangleright x' = x, Y \triangleright y' = y\}}
\end{array}$$

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

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 WF-VAR). If the path p has a module type exporting the type T, then $p.T$ is a well-formed type (rule WF-PATH).

A module type is well-formed, provided the set of its declarations is well-formed in the environment extended with themselves (rule WF-MODULE), and provided that dependencies between type definitions (including the nested ones) are not cyclic. Formally, its *static dependency graph* $\rightarrowtriangle_{(O,\emptyset)}$ is required to be acyclic. By rule WF-MIXIN, a mixin module type $\langle I; O; \to; \rightarrowtriangle\rangle$ 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 $\to$ is correct. A dynamic graph is said *correct* if its transitive closure is a partial ordering. The transitive closure of a dynamic graph $\to$ is defined in figure 5.3, as the set of paths of $\to$, labeled with the last edge of the path. This notion of graph correctness has been proven in chapter 4 to be a correct criterium 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 WF-MANIFEST), but the abstract type $\star$ is not a well-formed type by itself. Rule WF-ABSTR allows a declaration to use the abstract type, but only for type declarations. Notice that the abstract type $\star$ 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

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathit{SFV}(\star) & = & \emptyset & \mathit{SFV}(T \triangleright t : M) & = & \mathit{FV}(M) \\
\mathit{SFV}(t) & = & \emptyset & \mathit{SFV}(X \triangleright x : M) & = & \mathit{SFV}(M) \\
\mathit{SFV}(x.p.T) & = & \emptyset & & & \\
\mathit{SFV}(\{O\}) & = & \displaystyle\bigcup_{D \in O} \mathit{SFV}(D) \setminus \mathit{DV}(O) & & & \\
\mathit{SFV}(\langle I; O; \rightarrow; \rightarrowtriangle \rangle) & = & \displaystyle\bigcup_{D \in I} \mathit{SFV}(D) \setminus \mathit{DV}(I) \cup & & & \\
& & \displaystyle\bigcup_{D \in O} \mathit{SFV}(D) \setminus \mathit{DV}(I, O) & & &
\end{array}$$

Static dependency graph

$$\frac{\mathit{Node}(D_{i_1}) \rightarrowtriangle_{(\{D_1, D_2 \ldots D_n\}, \rightarrow)} \mathit{Node}(D_{i_2}) \qquad \mathit{Node}(D_{i_2}) \rightarrow \mathit{Node}(D_{i_3})}{\mathit{Node}(D_{i_1}) \rightarrowtriangle_{(\{D_1, D_2 \ldots D_n\}, \rightarrow)} \mathit{Node}(D_{i_3})}$$

$$\frac{\mathit{Node}(D_{i_1}) \rightarrow \mathit{Node}(D_{i_2}) \qquad \mathit{Node}(D_{i_2}) \rightarrowtriangle_{(\{D_1, D_2 \ldots D_n\}, \rightarrow)} \mathit{Node}(D_{i_3})}{\mathit{Node}(D_{i_1}) \rightarrowtriangle_{(\{D_1, D_2 \ldots D_n\}, \rightarrow)} \mathit{Node}(D_{i_3})}$$

$$\frac{s \in \mathit{SFV}(D_{i_2}) \qquad s = \mathit{DV}(D_{i_1})}{\mathit{Node}(D_{i_1}) \rightarrowtriangle_{(\{D_1, D_2 \ldots D_n\}, \rightarrow)} \mathit{Node}(D_{i_2})}$$

Figure 5.2: Static dependencies in a signature

$$\frac{X \xrightarrow{\chi_1}{}^{+} Z \qquad Z \xrightarrow{\chi_2} Y}{X \xrightarrow{\chi_2}{}^{+} Y} \qquad\qquad \frac{X \xrightarrow{\chi} Y}{X \xrightarrow{\chi}{}^{+} Y}$$

Figure 5.3: Transitive closure of →

Types

$$\frac{t \in DV(\Gamma)}{\Gamma \vdash t}\ (\textsc{Wf-Var}) \qquad \frac{\Gamma \vdash p : \{O\} \qquad T \in DN(O)}{\Gamma \vdash p.T}\ (\textsc{Wf-Path})$$

$$\frac{\Gamma + O \vdash O \qquad \vdash \rightarrowtriangle_{(O,\emptyset)}}{\Gamma \vdash \{O\}}\ (\textsc{Wf-Module})$$

$$\frac{\Gamma + I \vdash I \qquad \Gamma + I + O \vdash O \qquad DV(I) \perp DV(O) \qquad \vdash \rightarrowtriangle_{(I \cup O,\emptyset)} \qquad \vdash \to}{\Gamma \vdash \langle I; O; \to; \rightarrowtriangle \rangle}\ (\textsc{Wf-Mixin})$$

Declarations and signatures

$$\Gamma \vdash T \triangleright t : \star\ (\textsc{Wf-Abstr}) \qquad \frac{\Gamma \vdash M}{\Gamma \vdash S \triangleright s : M}\ (\textsc{Wf-Manifest})$$

$$\frac{\begin{array}{c}\forall D \in O, \Gamma \vdash D \\ \forall D, D' \in O (DN(D) = DN(D') \vee DV(D) = DV(D')) \implies D = D'\end{array}}{\Gamma \vdash O}\ (\textsc{Wf-Sig})$$

Figure 5.4: Well-formedness

$$\begin{array}{lcl}
\{D_1 \ldots D_n\}/p & = & \{D_1/p \ldots D_n/p\} \\
M/p & = & M \text{ (otherwise)} \\[1ex]
(T \triangleright t : \star)/p & = & T \triangleright t : p.T \\
(T \triangleright t : M)/p & = & T \triangleright t : M \text{ (otherwise)} \\
(X \triangleright x : M)/p & = & X \triangleright x : (M/p.X)
\end{array}$$

Figure 5.5: Type strengthening

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 $\{T \triangleright t : \star\}$, 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

$$\lceil O \mapsto p.O \rceil = \{s \mapsto p.S \mid (S, s) \in dom(O)\}.$$

Dynamic free variables

$$
\begin{array}{lcl}
\mathit{DFV}(x) & = & \{x\} \\
\mathit{DFV}(x.p.X) & = & \{x\} \\
\mathit{DFV}(\{o_v\}) & = & \bigcup\limits_{d \in o_v} \mathit{DFV}(d) \setminus \mathit{DV}(o_v) \\
\mathit{DFV}(\mathsf{let\ rec}\ o\ \mathsf{in}\ e) & = & (\bigcup\limits_{d \in o} \mathit{DFV}(d) \cup \mathit{DFV}(e)) \setminus \mathit{DV}(o) \\
\mathit{DFV}(\langle I; o\rangle) & = & (\bigcup\limits_{d \in o} \mathit{DFV}(d)) \setminus \mathit{DV}(o) \setminus \mathit{DV}(I) \\
\mathit{DFV}(e_1 + e_2) & = & \mathit{DFV}(e_1) \cup \mathit{DFV}(e_2) \\
\mathit{DFV}(\mathsf{close}\, e) & = & \mathit{DFV}(e) \\
\mathit{DFV}((e : M)) & = & \mathit{DFV}(e)
\end{array}
$$

Dynamic dependency graph

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

$$
\frac{(L_i, x_i) \in \mathit{dom}(I) \cup \mathit{dom}(o) \qquad (L[x_1 \ldots x_n] \triangleright x = e) \in o}{\mathit{Node}(L_i, x_i) \xrightarrow{\frownie}_{\langle I; o\rangle} \mathit{Node}(L, x)}
$$

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 Γ_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 Γ_o, yielding a signature O_o. But O_o still might depend on the anonymous definitions. The type 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 $\rightarrowtriangle_{(I \cup \Gamma_o, \rightarrow_{\langle I;o\rangle})}$ is checked acyclic; and the *concrete dependency graph* of $\langle I; o\rangle$, denoted by $\rightarrow_{\langle I;o\rangle}$, 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 $\mathit{Vars} \cup \mathit{Names}$. 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 $(L_i, x_i) \in \mathit{dom}(o) \cup \mathit{dom}(I)$, then there is an edge $\mathit{Node}(L_i, x_i) \xrightarrow{\frownie} \mathit{Node}(L, x)$. Second, if the body e of a definition $d = (L \triangleright x[z_1 \ldots z_n] = e)$ dynamically depends on a variable x', such that $(L', x') \in \mathit{dom}(I) \cup \mathit{dom}(o)$, then there is an edge from $\mathit{Node}(L', x')$ to $\mathit{Node}(L, x)$. The notion of dynamic dependence is defined in figure 5.6, and roughly corresponds to forgetting type constraints. The degree of the edge is $\mathit{Degree}(x', e)$, where the Degree function is defined for $x \in \mathit{DFV}(e)$ by

$$
\begin{array}{rcl}
\mathit{Degree}(x, \langle I; o\rangle) & = & \smiley \\
\mathit{Degree}(x, \{o_v\}) & = & \smiley \\
\mathit{Degree}(x, e) & = & \frownie \text{ otherwise.}
\end{array}
$$

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

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

Lift

$$\lfloor \rightarrow \rfloor \quad = \quad \rightarrow^{\Box}|_{\mathit{Names} \times \mathit{Names}}$$

Figure 5.7: Lifting concrete dependency graphs

$$(I_1, O_1) \Bumpeq (I_2, O_2) \text{ means } \begin{cases} I_1 \sqsubset_{O_2} I_2 \text{ and} \\ I_2 \sqsubset_{O_1} I_1 . \end{cases}$$

$I_1 \sqsubset_O I_2$ means that for all $(L, s) \in \mathit{dom}(I_1)$,
$x \in \mathit{FV}(I_2, O) \cup \mathit{DV}(I_2, O) \Rightarrow (L, s) \in \mathit{dom}(I_2)$ and $L \in \mathit{Names}$.

Figure 5.8: Compatibility

Rule TT-Module types basic modules $\{o_v\}$, 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 $\langle I_i; O_i; \rightarrow_i; \rightarrowtriangle_i \rangle$, 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 $\langle I; O; \rightarrow; \rightarrowtriangle \rangle$ is instantiated as follows. Semantically, the variables defined by O must replace the input variables of I. This is done by the substitution σ, 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 $\langle \emptyset; X \triangleright x = \{\} \rangle + (\langle \emptyset; X \triangleright x = \{\} \rangle : \langle \emptyset; \emptyset; \emptyset; \emptyset \rangle)$ 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

$$\frac{\Gamma(x) = M}{\Gamma \vdash x : M/x}\ (\text{TT-}\textsc{Var}) \qquad\qquad \frac{\Gamma \vdash p : \{O\}}{\Gamma \vdash p.X : O(X)\lceil O \mapsto p.O \rceil}\ (\text{TT-}\textsc{Access})$$

$$\frac{\begin{array}{c}\Gamma + I \vdash I \qquad \Gamma + I + O \vdash O \qquad \vdash \rightarrow_{\langle I;o \rangle} \qquad \vdash \rightarrowtriangle_{(I \cup \Gamma_o, \rightarrow_{\langle I;o \rangle})} \\ \Gamma + I + \Gamma_o \vdash \Gamma_o \qquad \Gamma + I + \Gamma_o \vdash o : \Gamma_o \qquad \Gamma + I + \Gamma_o \vdash \Gamma_{o|\,Names} \leq O\end{array}}{\Gamma \vdash \langle I;o \rangle : \langle I; O; \lfloor \rightarrow_{\langle I;o \rangle} \rfloor; (\rightarrowtriangle_{(I \cup \Gamma_o, \rightarrow_{\langle I;o \rangle})})^{+}{}_{|\,Names} \rangle}\ (\text{TT-}\textsc{Struct})$$

$$\frac{\Gamma \vdash o_v : \Gamma_o \qquad \Gamma + O \vdash \Gamma_{o|\,Names} \leq O \qquad \Gamma + O \vdash O \qquad \vdash \rightarrowtriangle_{(O,\emptyset)}}{\Gamma \vdash \{o_v\} : \{O\}}\ (\text{TT-}\textsc{Module})$$

$$\frac{\begin{array}{c}\Gamma + \Gamma_o \vdash \Gamma_o \qquad \Gamma \vdash M \qquad \Gamma + \Gamma_o \vdash o : \Gamma_o \\ \vdash \rightarrow_{\langle \emptyset;o \rangle} \qquad \vdash \rightarrowtriangle_{(\Gamma_o, \rightarrow_{\langle \emptyset;o \rangle})} \qquad \Gamma + \Gamma_o \vdash e : M' \qquad \Gamma + \Gamma_o \vdash M' \leq M\end{array}}{\Gamma \vdash \mathsf{let\ rec}\ o\ \mathsf{in}\ e : M}\ (\text{TT-}\textsc{Letrec})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \langle I_1; O_2; \rightarrow_1; \rightarrowtriangle_1 \rangle \qquad \Gamma \vdash e_2 : \langle I_2; O_2; \rightarrow_2; \rightarrowtriangle_2 \rangle \\ (I_1, O_1) \asymp (I_2, O_2) \qquad \vdash (\rightarrow_1 \cup \rightarrow_2) \qquad \Gamma + I \vdash I \\ \Gamma + I \vdash I \leq I_1 \qquad \Gamma + I \vdash I \leq I_2 \qquad \vdash (\rightarrowtriangle_1 \cup \rightarrowtriangle_2 \cup \rightarrowtriangle_{(I,(\rightarrow_1 \cup \rightarrow_2))})\end{array}}{\Gamma \vdash e_1 + e_2 : \langle I; O_1 + O_2; \rightarrow_1 \cup \rightarrow_2; \rightarrowtriangle_1 \cup \rightarrowtriangle_2 \cup \rightarrowtriangle_{(I,(\rightarrow_1 \cup \rightarrow_2))} \rangle}\ (\text{TT-}\textsc{Composition})$$

$$\frac{\Gamma \vdash e : M' \qquad \Gamma \vdash M \qquad \Gamma \vdash M' \leq M}{\Gamma \vdash (e : M) : M}\ (\text{TT-}\textsc{Constraint})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \langle I; O; \rightarrow; \rightarrowtriangle \rangle \qquad \sigma = \{s \mapsto s' \mid (S,s) \in dom(I), (S,s') \in dom(O)\} \\ I' = I\{\sigma\} \qquad O' = O\{\sigma\} \qquad \Gamma + O' \vdash \{O'\} \leq \{I'\}\end{array}}{\Gamma \vdash \mathsf{close}\, e : \{O'\}}\ (\text{TT-}\textsc{Close})$$

$$\frac{\forall 1 \leq i \leq n, \Gamma \vdash d_i : D_i}{\Gamma \vdash (d_1 \ldots d_n) : (D_1 \ldots D_n)}\ (\text{TT-}\textsc{Output}) \qquad \frac{\Gamma \vdash e : M}{\Gamma \vdash (X \triangleright x[y^*] = e) : (X \triangleright x : M)}\ (\text{TT-}\textsc{Expr})$$

$$\frac{\Gamma \vdash M}{\Gamma \vdash (T \triangleright t : M) : (T \triangleright t : M)}\ (\text{TT-}\textsc{Type})$$

Figure 5.9: Typing rules

Types

$$\frac{\Gamma \vdash M_1 \cong M_2}{\Gamma \vdash M_1 \leq M_2} \quad (\text{ST-Equiv}) \qquad \frac{\Gamma + O_1 + O_1' \vdash O_1 \leq O_2}{\Gamma \vdash \{O_1 + O_1'\} \leq \{O_2\}} \quad (\text{ST-Module})$$

$$\frac{\begin{array}{c}\Gamma + I_2 + I_2' \vdash I_2 \leq I_1 \\ \Gamma + I_2 + I_2' + O_1 \vdash O_1 \leq O_2 \qquad \rightarrow_1 \leq \rightarrow_2 \qquad \rightarrowtriangle_1 \subset \rightarrowtriangle_2\end{array}}{\Gamma \vdash \langle I_1; O_1; \rightarrow_1; \rightarrowtriangle_1\rangle \leq \langle I_2 + I_2'; O_2; \rightarrow_2; \rightarrowtriangle_2\rangle} \quad (\text{ST-Mixin})$$

Signatures

$$\frac{\forall 1 \leq i \leq n, \Gamma \vdash D_i \leq D_i'}{\Gamma \vdash D_1 \ldots D_n \leq D_1' \ldots D_n'} \quad (\text{ST-Sig})$$

Declarations

$$\frac{\Gamma \vdash M_1 \leq M_2}{\Gamma \vdash (X \triangleright x : M_1) \leq (X \triangleright x : M_2)} \quad (\text{ST-Val}) \qquad \Gamma \vdash (T \triangleright t : M) \leq (T \triangleright t : \star) \quad (\text{ST-Con-Abs})$$

$$\frac{\Gamma \vdash t \cong M}{\Gamma \vdash (T \triangleright t : \star) \leq (T \triangleright t : M)} \quad (\text{ST-Abs-Con}) \qquad \Gamma \vdash (T \triangleright t : \star) \leq (T \triangleright t : \star) \quad (\text{ST-Abs-Abs})$$

$$\frac{\Gamma \vdash M_1 \cong M_2}{\Gamma \vdash (T \triangleright t : M_1) \leq (T \triangleright t : M_2)} \quad (\text{ST-Con-Con})$$

Figure 5.10: Subtyping and signature matching

$$\begin{array}{llr}(\langle\emptyset; & X \triangleright x = \{Y \triangleright y = x'\}, & \leftarrow \langle\emptyset; X \triangleright x = \{\}\rangle \\ & \_ \triangleright z = x.Y & \rangle : \langle\emptyset;\emptyset;\emptyset;\emptyset\rangle)\end{array}$$

which reduces to the ill-typed $\langle\emptyset; X \triangleright x = \{\}, \_ \triangleright z = x.Y\rangle$

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 possibly 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 its input signature is less precise, i.e. it 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 ☺ labels into ☹ 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 $T \triangleright t : \star$ 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 $\{T \triangleright t : \star, U \triangleright u : t\}$ and $\{U \triangleright u : \star, T \triangleright t : u\}$. This leads to comparing the declarations $T \triangleright t : \star$ and $T \triangleright t : u$ in the environment $T \triangleright t : \star, U \triangleright u : t$, where u is provably equal to t.

Types

$$\frac{M \neq \star \qquad \Gamma \vdash M}{\Gamma \vdash M \cong M} \ (\text{TE-Refl}) \qquad \frac{\Gamma(t) \neq \star}{\Gamma \vdash t \cong \Gamma(t)} \ (\text{TE-Var})$$

$$\frac{\Gamma \vdash p : \{O\} \qquad O(T) \neq \star}{\Gamma \vdash p.T \cong O(T)\lceil O \mapsto p.O \rceil} \ (\text{TE-Acc}) \qquad \frac{\Gamma + O_1 \vdash O_1 \cong O_2}{\Gamma \vdash \{O_1\} \cong \{O_2\}} \ (\text{TE-Module})$$

$$\frac{\Gamma + I_2 \vdash I_2 \cong I_1 \qquad \Gamma + I_2 \vdash O_1 \cong O_2 \qquad \rightarrow_1 = \rightarrow_2 \qquad \rightarrowtriangle_1 = \rightarrowtriangle_2}{\Gamma \vdash \langle I_1; O_1; \rightarrow_1; \rightarrowtriangle_1 \rangle \cong \langle I_2; O_2; \rightarrow_2; \rightarrowtriangle_2 \rangle} \ (\text{TE-Mixin})$$

Signatures

$$\frac{\forall 1 \leq i \leq n, \Gamma \vdash D_i \cong D'_i}{\Gamma \vdash \{D_1 \ldots D_n\} \cong \{D'_1 \ldots D'_n\}} \ (\text{TE-Sig})$$

Declarations

$$\frac{\Gamma \vdash M_1 \cong M_2}{\Gamma \vdash (S \triangleright s : M_1) \cong (S \triangleright s : M_2)} \ (\text{TE-Comp})$$

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

```
module A = (struct
  type t = int
  let x = 1
end : sig
  type t
  val x : t
end)
```

(Here, the construction `(module : module-type)` denotes the coercion of a module to a module type.)

The module `A` is bound in the typing environment with the type

```
sig
  type t
  val x :  t
end
```

Now, if further in the program we use `A.x`, then its type is simply `A.t`, not `int`. Indeed, in the type of `A`, no definition is provided for `t`. Until now, no difficulty arose, but if we try to evaluate our program, then `A.x` evaluates to `1`, which is of type `int`, but not of type `A.t`: type preservation does not hold.

Lillibridge's solution Lillibridge [56] defines a kernel module system called the *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 m $=_{\text{def}}$

```
struct
 type t = int
 let x = 1
end
```

and S $=_{\text{def}}$

```
sig
 type t
 val x : t
end
```

The counter-example program showing that type preservation does not hold in the manifest types approach is `module A = (`m` : `S`), let res = A.x`. Recall that in the manifest types approach, this expression is well-typed, and `res` has type `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 `(`m `:` S`)` is known to be S: `(functor (A : `S`) = struct let res = A.x end)` m.

In Lillibridge's system, this expression is well-typed. Indeed, m has type `sig type t = int val x : int end`. Moreover, the functor has type `functor (A : sig type t val x : t end) -> sig val res : A.t end` as a principal type. This type is a subtype of `functor (A : sig type t = int val x : int end) -> 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 `int`. Therefore, when selection is performed, this type is preserved.

However, one could object that we cheated a bit here, by forgetting that m was initially coerced to S. And indeed, if we replace m with m `:` S in our encoding, we obtain an ill-typed expression. Indeed, the principal type of the argument to the functor, m `:` S, is 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.

Related approaches 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.

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 Lillibridge'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 Lillibridge's thesis for more details.

```
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

$$\begin{array}{l}\{\ \mathsf{type}\, T \triangleright t : \star, \\ \quad \mathsf{val}\, X \triangleright x : \{\, \mathsf{type}\, T \triangleright t' = t\}\ \}\end{array}$$

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.

	Notation	Denotation
Function	$\lambda x : M.e$	$\langle ARG \triangleright x : M; RES \triangleright res = e \rangle$
Application	$e_1 e_2$	$\mathsf{let\ rec}\ res = \mathsf{close}(e_1 + \langle \emptyset; ARG \triangleright arg = e_2 \rangle)$ $\mathsf{in}\ res.RES$
Function type	$M_1 \to M_2$	$\langle\ ARG \triangleright arg : M_1; \quad \{ARG \overset{☹}{\longrightarrow} RES\};\ \rangle$ $RES \triangleright res : M_2; \quad \{ARG \rightarrowtriangle RES\}$
Type function	$\Lambda t.e$	$\langle TARG \triangleright t : \star; RES \triangleright res = e \rangle$
Type application	$e[M]$	$\mathsf{let\ rec}\ res = \mathsf{close}(e_1 + \langle \emptyset; TARG \triangleright targ = M \rangle)$ $\mathsf{in}\ res.RES$
Type function type	$\forall t.M$	$\langle TARG \triangleright t : \star; RES \triangleright res : M; \emptyset; \{TARG \rightarrowtriangle RES\} \rangle$

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.

Inuitively, 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

$S_1 \quad =_{\text{def}}$

```
sig
 type u = A of u * u | B of int
 type t = u * u
end
```

is interpreted as

$S_1^{opaque} \quad =_{\text{def}}$

```
sig
 type u
 type t = u * u
 val u_in : (u * u + int) -> u
 val u_out : u -> (u * u + int)
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

$S_1^{transparent}$ $=_{\mathrm{def}}$

```
sig
 type u = μ u . (u * u) + int
 type t = u * u
end
```

Notice that there is no need for introducing special constructors, as `u_in` and `u_out` in the opaque interpretation, since one can rely on the sum type injections to produce values of type `u`. Furthermore, the recursive type constructor μ is difficult to deal with. In their papers on recursive modules [27, 29], Harper et al. study two possible type theoretic constructions implementing μ, distinguishing *equi-recursive* types from *iso-recursive* types.

In the equi-recursive approach, the type `τ = μ u . (u * u) + int` above is equivalent to its unrolling `(τ * τ) + int`. To construct a value of type τ, construct a value of type `int` or `int * τ`, and just inject it into the sum type, thanks to the left and right injections injl and injr, respectively. For example, $e =_{\mathrm{def}} \mathsf{injr}\, 1$ has type τ. Such expressions are decomposed by the projection operations of sum types, projl and projr, so one can recover the integer from e by $\mathsf{projr}\, e$.

In the iso-recursive approach, τ is only isomorphic to its unrolling `(τ * τ) + int`. Concretely, it means that given some term e' of type `(τ * τ) + int`, there is a rolling operation roll that coerces e' to τ: $\mathsf{roll}\, e'$ is of type τ. Conversely, to use a value of type τ, one has to apply the unroll operation first, which coerces it to $(\tau * \tau) + int)$. For instance, to construct a value of type τ, one writes $e =_{\mathrm{def}} \mathsf{roll}(\mathsf{injr}\, 1)$, and its first element is accessed through $(\mathsf{unroll}(\mathsf{projr}\, e))$.

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 $=_{\mathrm{def}}$

```
sig
 type t
 type u = A of t | B of int
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 = τ` and `t = u * u`, the type τ is equivalent to `t + int`. It is possible, by replacing `u` with its value, to prove that `t` is equivalent to `μ u . (τ * τ) + int`, but this type is not equivalent to τ. In order to solve the problem, Crary et al. enrich the type system with *Shao's equation*:

$$\mu\alpha.\tau = \mu\alpha.(\tau\{\alpha \mapsto (\mu\alpha.\tau)\}) \quad (\textsc{Shao})$$

This allows to recover Standard ML datatypes.

We choose yet another approach, closer to *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 *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 *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 $C \in \mathit{ConNames}$. 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 $C^{pt}[e_1 \dots e_n]$, 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 $\mathsf{match}^{pt}_{\Phi}$. It is indexed by a type path pt, and a datatype Φ. 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 Φ. A datatype $\Phi = \phi_1 \dots \phi_n$ is a list of constructor definitions, syntactically required not to bind the same constructor name twice, and a constructor definition $\phi = C[M_1 \dots M_n]$ 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 $T \triangleright t = \star$, while the other specifies its constructors $t \prec \Phi$.

The set of expressions of syntactically predictable shape is extended with constructor applications $C^{pt}[e_1 \dots e_n]$, 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 ☺, if it is ☺ 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 $\mathsf{match}^{pt}_{\Phi}$ expects the argument to the matching, plus $\mid \Phi \mid$ functions for dealing with each of the constructors defined by Φ. 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 $\mathsf{match}^{pt}_{\Phi}$, and the first argument to the matching is ${C_i}^{pt'}[v'_1 \dots v'_{n_i}]$, according to the index of C_i in Φ, the rule applies one of the matching functions $v_1 \dots v_n$ to the arguments $v'_1 \dots v'_{n_i}$.

Static semantics As shown by figure 5.14, the static semantics of *MML* is extended to account for datatypes. A new judgment, *type unfolding* $\prec$, 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 $p.T$, then the datatype has to be extracted from the type of p.

Typing constructor application $C^{pt}[e_1 \dots e_n]$ (rule TT-CONAPP) consists in unfolding the type path annotation pt, to retreive the corresponding datatype Φ, and check that the arguments match the types expected by Φ. Typing a matching operator $\mathsf{match}^{pt}_{\Phi'}$ is a bit more complicated. There are two main checks to do: first, the pt annotation must unfold to a datatype Φ, and second the Φ' annotation must be equivalent to that Φ. Then, the type of $\mathsf{match}^{pt}_{\Phi'}$ is a polymorphic type $\forall t.M$, where M is a function expecting the first argument of type pt, plus the $\mid \Phi \mid$ matching

Syntax

Type path: $pt ::= t \mid p.T$

Expression: $e ::= \ldots \mid C^{pt}[e_1 \ldots e_n] \mid \mathsf{match}^{pt}_{\Phi}$

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

$e_{\downarrow} ::= \ldots \mid C^{pt}[e_1 \ldots e_n]$

Static free variables

$SFV(T \prec \Phi) = \emptyset$

Well-formedness

$$\frac{\forall i \in \{1 \ldots n\}, \forall j \in \{1 \ldots n_i\}, \Gamma \vdash M^i_j \qquad \forall i, j \in \{1 \ldots n\}, \Gamma \vdash C_i \neq C_j}{\Gamma \vdash (C_1[M^1_1 \ldots M^1_{n_1}] \ldots C_n[M^n_1 \ldots M^n_{n_n}])} \quad (\textsc{Wf-Datatype})$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash \Phi}{\Gamma \vdash (t \prec \Phi)} \quad (\textsc{Wf-Unfold})$$

$$\frac{\begin{array}{c}\forall D \in O, \Gamma \vdash D \qquad \forall (t \prec \Phi), (t \prec \Phi') \in O, \Phi = \Phi' \\ \forall (t \prec \Phi), (T \triangleright t : M) \in O, (M \equiv \star) \vee ((\Gamma \vdash M \prec \Phi') \wedge (\Gamma \vdash \Phi \cong \Phi')) \\ \forall D, D' \in O(DN(D) = DN(D') \vee DV(D) = DV(D')) \implies D = D'\end{array}}{\Gamma \vdash O} \quad (\textsc{Wf-Sig'})$$

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: $v ::= \ldots \mid C^{pt}[v_1 \ldots v_n]$ Constructed value
$\mid (\mathsf{match}^{pt}_{\Phi}\, v_1 \ldots v_n)$ Partial matching (for $n \leq\mid \Phi \mid$)

$$\frac{\Phi = (C_1[M^1_1 \ldots M^1_{n_1}] \ldots C_n[M^n_1 \ldots M^n_{n_n}])}{\mathsf{match}^{pt}_{\Phi}({C_i}^{pt'}[v'_1 \ldots v'_{n_i}]) v_1 \ldots v_n \longrightarrow (v_i v'_1 \ldots v'_{n_i})} \quad (\textsc{Match})$$

Figure 5.13: Extending the dynamic semantics

functions, and returning a value of type t. The matching function corresponding to the constructor $C[M_1 \dots M_n]$ expects n arguments of types $M_1 \dots M_n$, 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 type and 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 *not* and infix and operators, and a conditional construction if then else .

5.4.1 Lists

We program a simple module implementing lists in *MML*. If we stick to monomorphic lists, that is, the type of elements is fixed to *int* for example, then it is straighforward. Let $\Phi = Nil, Cons[int, t]$. We define the module *list* by

$$\begin{array}{lll}
o_{list} =_{\text{def}} & \mathsf{type}\, T \triangleright t = \Phi & \\
& \mathsf{val}\, Head \triangleright head = \lambda x.\, \mathsf{match}^{t}_{\Phi}[int]\, x & error \\
& & \lambda hd \lambda tl.hd \\
& \mathsf{val}\, Tail \triangleright tail = \lambda x.\, \mathsf{match}^{t}_{\Phi}[t]\, x & error \\
& & \lambda hd \lambda tl.tl \\
& \mathsf{val}\, Map \triangleright map = \lambda f \lambda x.\, \mathsf{match}^{t}_{\Phi}[t]\, x & Nil^{t}[] \\
& & \lambda hd \lambda tl. Cons^{t}[(f\ hd), (map\ f\ tl)]
\end{array}$$

and $list = \mathsf{close}\, \langle \emptyset; o_{list} \rangle$.

Notice the use of *error*: we did not include exceptions in our formalism, but for sure they remain a useful construction in programming, and should be included in any practical application. The obtained module is of type $\{O\}$, where

$$\begin{array}{ll}
O =_{\text{def}} & \mathsf{type}\, T \triangleright t : \star \\
& t \prec Nil, Cons[int, t] \\
& \mathsf{val}\, Head \triangleright head : t \to int \\
& \mathsf{val}\, Tail \triangleright tail : t \to t \\
& \mathsf{val}\, Map \triangleright map : (int \to int) \to t \to t
\end{array}$$

Type path unfolding $\Gamma \vdash pt \prec \Phi$

$$\frac{(t \prec \Phi) \in \Gamma}{\Gamma \vdash t \prec \Phi}\ (\textsc{TU-Var}) \qquad \frac{\Gamma \vdash p : \{O\} \qquad (T \triangleright t : M), (t \prec \Phi) \in O}{\Gamma \vdash p.T \prec \Phi\lceil O \mapsto p.O\rceil}\ (\textsc{TU-Path})$$

$$\frac{\Gamma \vdash pt \cong M \qquad \Gamma \vdash M \prec \Phi}{\Gamma \vdash pt \prec \Phi}\ (\textsc{TU-Eq})$$

Expressions

$$\frac{\Gamma \vdash pt \prec \Phi \qquad (C[M_1 \ldots M_n]) \in \Phi \qquad \forall i \in \{1 \ldots n\}, \Gamma \vdash e_i : M_i}{\Gamma \vdash C^{pt}[e_1 \ldots e_n] : pt}\ (\textsc{TT-ConApp})$$

$$\frac{\Gamma \vdash pt \prec \Phi \qquad \Gamma \vdash \Phi \cong \Phi'}{\Gamma \vdash \mathsf{match}^{pt}_{\Phi'} : \mathit{Match}(pt, \Phi)}\ (\textsc{TT-Match})$$

Definitions

$$\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash d_i : O_i}{\Gamma \vdash (d_1 \ldots d_n) : (O_1 + \ldots + O_n)}\ (\textsc{TT-Output'})$$

$$\frac{\Gamma \vdash \Phi}{\Gamma \vdash (T \triangleright t = \Phi) : (T \triangleright t : \star, t \prec \Phi)}\ (\textsc{TT-Datatype})$$

Declaration matching

$$\frac{\Gamma \vdash \Phi \cong \Phi'}{\Gamma \vdash (t \prec \Phi) \leq (t \prec \Phi')}\ (\textsc{ST-Datatype})$$

Signature matching

$$\frac{\forall 1 \leq i \leq n, \Gamma \vdash D_i \leq D'_i}{\Gamma \vdash D_1 \ldots D_n, (t \prec \Phi)^* \leq D'_1 \ldots D'_n}\ (\textsc{ST-Sig'})$$

Datatype equivalence

$$\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash M_i \cong M'_i}{\Gamma \vdash C[M_1 \ldots M_n] \cong C[M'_1 \ldots M'_n]}\ (\textsc{DE-Con})$$

$$\frac{\forall i \in \{1 \ldots n\}, \Gamma \vdash \phi_i \cong \phi'_i}{\Gamma \vdash (\phi_1 \ldots \phi_n) \cong (\phi'_1 \ldots \phi'_n)}\ (\textsc{DE-Datatype})$$

Type of $\mathsf{match}^{pt}_{\Phi}$

$$\begin{array}{lcl} \mathit{Match}(pt, \phi_1 \ldots \phi_n) & = & \forall t.pt \to \mathit{Constr}(t, \phi_1) \to \ldots \to \mathit{Constr}(t, \phi_n) \to t \\ \mathit{Constr}(t, C[M_1 \ldots M_n]) & = & (M_1 \to \ldots \to M_n \to t) \end{array}$$

Figure 5.14: Extension of the typing judments

Moreover, by type strengthening, at each place of use, the type declaration $\mathsf{type}\, T \triangleright t : \star$ of *list* becomes $\mathsf{type}\, T \triangleright t : \mathit{list}.T$.

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 type *elt* for the elements of the list. The corresponding mixin module *openList* has the input signature $I =_{\text{def}} \mathsf{type}\, \mathit{Elt} \triangleright \mathit{elt} : \star$ and the output is o_{list}, except that in the definition of T, *int* is replaced with *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 *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 Szyperski 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 Russo'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

$$\begin{array}{rcl} pt & ::= & t \mid p.T \\ & \mid & [p_1 + \ldots + p_n].T \end{array}$$

where the production $[p_1 + \ldots p_2].T$ would denote the type component T in any module computed by closing the sum $p_1 + \ldots + p_n$. Then, a type M can be encapsulated in a mixin module $\mathit{eltMix} = \langle \epsilon; \mathit{Elt} \triangleright \mathit{elt} = M \rangle$, and the type of lists with elements of type M is denoted by $[\mathit{openList} + \mathit{eltMix}].T$. The set of operations over lists can be extended polymorphically, as shown for instance by the following definition of the traditional functions *fold_left*, applying a function successively to all the elements of a list, and *assoc*, looking for the element associated to a value in an association list. We denote by Φ_{List} the datatype $\mathit{Nil}, \mathit{Cons}\left[\begin{array}{l} [\mathit{openList} + \mathit{eltMix}].\mathit{Elt}, \\ [\mathit{openList} + \mathit{eltMix}].T \end{array}\right]$ and by pt the type path $[\mathit{openList} + \mathit{eltMix}].T$.

We define

$$\begin{array}{ll} \mathsf{let\ rec}\, \mathit{fold\_left} = & \Lambda t.\lambda \mathit{eltMix}.\lambda f.\lambda \mathit{init}.\lambda l.\, \mathsf{match}^{pt}_{\Phi_{List}}[t]l \\ & \quad \mathit{init} \\ & \quad \lambda hd.\lambda tl.(\mathit{fold\_left}\,[t]\,\mathit{eltMix}\,f\,(f\,\mathit{init}\,hd)\,tl) \end{array}$$

and

$$\begin{array}{ll} \mathsf{let\ rec}\, \mathit{assoc} = & \lambda \mathit{eltMix}\langle \emptyset; \begin{array}{l} \mathsf{type}\, \mathit{Fst} \triangleright \mathit{fst} : \star \\ \mathsf{type}\, \mathit{Snd} \triangleright \mathit{snd} : \star \\ \mathsf{type}\, \mathit{Elt} \triangleright \mathit{elt} : \mathit{fst} \times \mathit{snd} \end{array} ; \emptyset; \emptyset\rangle. \\ & \quad \lambda v.\lambda l.\, \mathsf{match}^{pt}_{\Phi_{List}}[[\mathit{eltMix}].\mathit{Snd}]\, l \\ & \quad \mathit{error} \\ & \quad \lambda hd.\lambda tl.\ \begin{array}{l} \mathsf{if}\ \ \mathit{fst}\, hd = v \\ \mathsf{then}\ \ \mathit{snd}\, hd \\ \mathsf{else}\ \ \mathit{assoc}\ \mathit{eltMix}\ v\ tl \end{array} \end{array}$$

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 $\mathsf{val}\, \mathit{List} \triangleright \mathit{list} = \langle \mathsf{type}\, \mathit{Elt} \triangleright \mathit{elt} : \star; \mathsf{type}\, T \triangleright t = \mathit{Nil}, \mathit{Cons}[\mathit{elt}, t]\rangle$ and the type $\mathit{list}(M)$ is $[\mathit{list} + \mathit{mixElt}].T$, for *mixElt* a named mixin exporting the type M. It is not obvious that this works in practice, because the argument *mixElt* has to be named. In theory, all types could be wrapped in in mixin modules as their unique component *Elt*, and referred to by the name of these mixin modules. For example, the module for lists and the *Map* function would look like :

$$\begin{array}{l}
\mathit{list} = \mathsf{close}\langle\emptyset;\ \mathsf{val}\ \mathit{List} \triangleright \mathit{list} = \ \langle\ \ \mathit{Elt} \triangleright \mathit{elt} : \star; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad T \triangleright t = \mathit{Nil}, \mathit{Cons}[\mathit{elt}, t]\rangle \\
\qquad\qquad\quad \mathsf{val}\ \mathit{Map} \triangleright \mathit{map} = \\
\qquad\qquad\qquad \lambda \mathit{mixElt} : \langle\emptyset; \mathit{Elt} \triangleright \mathit{elt} : \star; \emptyset; \emptyset\rangle. \\
\qquad\qquad\qquad \Lambda t'. \\
\qquad\qquad\qquad \lambda f : [\mathit{mixElt}].\mathit{Elt} \rightarrow t'. \\
\qquad\qquad\qquad \lambda l : [\mathit{mixElt} + \mathit{list}].T. \\
\qquad\qquad\qquad \mathsf{match}^{[\mathit{mixElt}+\mathit{list}].T}_{\mathit{Nil}, \mathit{Cons}[[\mathit{mixElt}+\mathit{list}].\mathit{Elt}, [\mathit{mixElt}+\mathit{list}].T]}[t'] \\
\qquad\qquad\qquad\quad \mathit{Nil}^{[\mathit{mixElt}+\mathit{list}].T}[] \\
\qquad\qquad\qquad\quad \lambda \mathit{hd}.\lambda \mathit{tl}.\mathit{Cons}^{[\mathit{mixElt}+\mathit{list}].T}[(f\ \mathit{hd}), (\mathit{map}\ \mathit{mixElt}\ [t']\ f\ \mathit{tl})] \quad \rangle
\end{array}$$

This example fails to type-check, at least if f is given the type $[\mathit{mixElt}].\mathit{Elt} \rightarrow t'$, since its argument has type $[\mathit{mixElt} + \mathit{list}].\mathit{Elt}$. 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 *list* has been defined, using them, with the following type:

$$\begin{array}{l}
\mathit{list} : \{\ \mathsf{type}\, T \triangleright t : \Lambda \mathit{elt}.\mathit{list}.T(\mathit{elt}) \\
\qquad\quad t[\mathit{elt}] \prec \mathit{Nil}, \mathit{Cons}[\mathit{elt}, t[\mathit{elt}]] \\
\qquad\quad \mathsf{val}\, \mathit{Head} \triangleright \mathit{head} : \forall \mathit{elt}.t[\mathit{elt}] \rightarrow \mathit{elt} \\
\qquad\quad \mathsf{val}\ \mathit{Tail} \triangleright \mathit{tail} : \forall \mathit{elt}.t[\mathit{elt}] \rightarrow t[\mathit{elt}] \\
\qquad\quad \mathsf{val}\, \mathit{Fold\_left} \triangleright \mathit{fold\_left} : \forall \mathit{elt}.\forall t'.(t' \rightarrow \mathit{elt} \rightarrow t') \rightarrow t' \rightarrow t[\mathit{elt}] \rightarrow t' \\
\qquad\quad \mathsf{val}\, \mathit{Mem} \triangleright \mathit{mem} : \forall \mathit{elt}.\mathit{elt} \rightarrow t[\mathit{elt}] \rightarrow \mathit{bool} \\
\qquad\quad \mathsf{val}\, \mathit{Max} \triangleright \mathit{max} : \forall \mathit{elt}.(\mathit{elt} \rightarrow \mathit{elt} \rightarrow \mathit{int}) \rightarrow t[\mathit{elt}] \rightarrow \mathit{elt} \\
\qquad\quad \mathsf{val}\, \mathit{Assoc} \triangleright \mathit{assoc} : \forall \mathit{fst}.\forall \mathit{snd}.\mathit{fst} \rightarrow t[\mathit{fst} \times \mathit{snd}] \rightarrow \mathit{snd} \quad \}
\end{array}$$

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 *openEval* is in charge of the basic operations. It imports the type *env* of environments, the type *binding* of bindings, the type *variable* for expression variables, the function *find_in_env*, which retrieves the value of a variable in an environment, and the function *bind*, which binds an expression to a variable in the environment.

$$\begin{array}{ll} I_{openEval} =_{\mathrm{def}} & \mathsf{type}\ \mathit{Env} \triangleright \mathit{env} : \star \\ & \mathsf{type}\ \mathit{Binding} \triangleright \mathit{binding} : \star \\ & \mathsf{type}\ \mathit{Result} \triangleright \mathit{result} : \mathit{int} \\ & \mathsf{type}\ \mathit{Variable} \triangleright \mathit{variable} : \star \\ & \mathsf{val}\ \mathit{Find\_in\_env} \triangleright \mathit{find\_in\_env} : \mathit{variable} \to \mathit{env} \to \mathit{result} \\ & \mathsf{val}\ \mathit{Bind} \triangleright \mathit{bind} : \mathit{bindings} \to \mathit{env} \to \mathit{env} \end{array}$$

The mixin *openEval* must the define the datatype *expr* of expressions, the type *result* for results of evaluation (integers), and the function *eval* which evaluates an expression in an environment. The datatype of expressions is defined as

$$\begin{array}{lll} \Phi_{Expr} =_{\mathrm{def}} & \mathit{Var}[\mathit{variable}], & \text{(* Variable *)} \\ & \mathit{Plus}[\mathit{expr}, \mathit{expr}], & \text{(* Addition *)} \\ & \mathit{Const}[\mathit{int}], & \text{(* Integer constant *)} \\ & \mathit{Let}[\mathit{binding}, \mathit{expr}] & \text{(* Let binding *)} \end{array}$$

The output of the mixin module is as follows:

$$\begin{array}{ll} o_{openEval} =_{\mathrm{def}} & \mathsf{type}\ \mathit{Expr} \triangleright \mathit{expr} = \Phi_{Expr} \\ & \mathsf{type}\ \mathit{Result} \triangleright \mathit{result}' : \mathit{int} \\ & \mathsf{val}\ \mathit{Eval} \triangleright \mathit{eval} = \lambda \mathit{an\_env}.\lambda \mathit{an\_expr}.\ \mathsf{match}^{\mathit{expr}}_{\Phi_{Expr}}\ [\mathit{result}]\ \mathit{an\_expr} \\ & \quad \lambda v.\mathit{find\_in\_env}\ v\ \mathit{an\_env} \\ & \quad \lambda \mathit{an\_expr}_1.\lambda \mathit{an\_expr}_2.(\mathit{eval}\ \mathit{an\_env}\ \mathit{an\_expr}_1) + (\mathit{eval}\ \mathit{an\_env}\ \mathit{an\_expr}_2) \\ & \quad \lambda n.n \\ & \quad \lambda \mathit{binding}.\lambda \mathit{an\_expr}.\mathit{eval}\ (\mathit{bind}\ \mathit{binding}\ \mathit{an\_env})\ \mathit{an\_expr} \end{array}$$

and we can define *openEval* by $\mathit{openEval} = \langle I_{openEval}; o_{openEval} \rangle$.
It has type $\langle I_{openEval}; O_{openEval}; \rightarrow_{openEval}; \rightarrowtriangle_{openEval} \rangle$, where

$$\begin{array}{lll} O_{openEval} & =_{\mathrm{def}} & \mathsf{type}\ \mathit{Expr} \triangleright \mathit{expr} : \star \\ & & \mathit{expr} \prec \Phi_{Expr} \\ & & \mathsf{type}\ \mathit{Result} \triangleright \mathit{result}' : \mathit{int} \\ & & \mathsf{val}\ \mathit{Eval} \triangleright \mathit{eval} : \mathit{env} \to \mathit{expr} \to \mathit{result} \\ \rightarrow_{openEval} & =_{\mathrm{def}} & \{\ \mathit{Eval} \overset{\smiley}{\longrightarrow} \mathit{Eval} \\ & & \quad \mathit{Find\_in\_env} \overset{\smiley}{\longrightarrow} \mathit{Eval} \\ & & \quad \mathit{Bind} \overset{\smiley}{\longrightarrow} \mathit{Eval}\ \} \\ \rightarrowtriangle_{openEval} & =_{\mathrm{def}} & \emptyset \end{array}$$

By rule ST-Sig', the implementation Φ_{idExpr} can be hidden, by type constraint.

Binding mixin module The second mixin module deals with bindings. It imports the types of environments, variables, expressions and results, and the functions *Eval* and *Add_to_env*, which adds a variable and its value to the environment. Its import signature is thus

$$\begin{array}{ll} I_{openBind} =_{\mathrm{def}} & \mathsf{type}\ \mathit{Env} \triangleright \mathit{env} : \star \\ & \mathsf{type}\ \mathit{Variable} \triangleright \mathit{variable} : \star \\ & \mathsf{type}\ \mathit{Expr} \triangleright \mathit{expr} : \star \\ & \mathsf{type}\ \mathit{Result} \triangleright \mathit{result} : \star \\ & \mathsf{val}\ \mathit{Add\_to\_env} \triangleright \mathit{add\_to\_env} : \mathit{variable} \to \mathit{result} \to \mathit{env} \to \mathit{env} \\ & \mathsf{val}\ \mathit{Eval} \triangleright \mathit{eval} : \mathit{env} \to \mathit{expr} \to \mathit{result} \end{array}$$

Given this, it can define the type *Binding* of bindings, as association lists of variables and expressions, and the function *bind* which takes a binding and an environment as arguments, evaluates the expressions, and binds the variables to the corresponding results in the environment.

$$
\begin{array}{lll}
o_{openBind} =_{\text{def}} & \mathsf{type}\ Binding \triangleright binding = list.T[variable \times expr] & \\
& \mathsf{val}\ Bind \triangleright bind = \lambda bindings.\lambda an\_env.(list.Fold\_left & [variable \times expr]\,[env] \\
& & bind\_one \\
& & an\_env \\
& & bindings) \\
& \mathsf{val}\ \_ \triangleright bind\_one = \lambda an\_env.\lambda pair.(add\_to\_env & (fst\ pair) \\
& & (eval\ an\_env\ (snd\ pair)) \\
& & an\_env)
\end{array}
$$

We can define the mixin module $openBind = \langle I_{openBind}; o_{openBind} \rangle$. The type *Binding* can be made abstract by type constraint, which gives *openBind* the type

$$openBind : \langle I_{openBind}; O_{openBind}; \rightarrow_{openBind}; \mathrel{-\!\!\triangleright}_{openBind} \rangle,$$

with

$$
\begin{array}{lll}
O_{openBind} & =_{\text{def}} & \mathsf{type}\ Binding \triangleright binding = \star \\
& & \mathsf{val}\ Bind \triangleright bind : bindings \rightarrow env \rightarrow env \\
\rightarrow_{openBind} & =_{\text{def}} & \{\ Add\_to\_env \overset{☺}{\longrightarrow} Bind \\
& & \quad Eval \overset{☺}{\longrightarrow} Bind \} \\
\mathrel{-\!\!\triangleright}_{openBind} & =_{\text{def}} & Variable \mathrel{-\!\!\triangleright} Binding \\
& & Expr \mathrel{-\!\!\triangleright} Binding
\end{array}
$$

Environment mixin module The last mixin module we define handles environments. It has to define the type *Env* of environments, and the functions *Find_in_env* and *Add_to_env* for finding and adding a variable binding in environments. It can be implemented by lists, as follows:

$$
\begin{array}{lll}
openEnv = & \langle & \mathsf{type}\ Variable \triangleright variable : \star \\
& & \mathsf{type}\ Result \triangleright result : \star \\
& & \mathsf{type}\ Env \triangleright env : list.T[variable \times result] \\
& ; & Find\_in\_env \triangleright find\_in\_env = list.Assoc\,[variable]\,[result] \\
& & Add\_to\_env \triangleright add\_to\_env = \lambda v.\lambda res.\lambda an\_env.Cons^{env}[(v, res), an\_env]\ \rangle
\end{array}
$$

Once again, the implementation of the type *Env* can be hidden to the outside world by type constraint. Finally, the interpreter module is obtained by $\mathsf{close}(openEval + openBind + openEnv)$.

Comparisons We think [?] that the example compiler sketched by Duggan and Sourelis in [31] is implementable in *MML* quite straighforwardly. 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], Crary, 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*.

$$
\begin{array}{lll}
openSet = & \langle & \mathsf{type}\ Elt \triangleright elt : \star \\
 & & \mathsf{val}\ Elt\_cmp \triangleright elt\_cmp : elt \to elt \to bool \\
 & ; & \mathsf{type}\ Set \triangleright set = list.T[elt] \\
 & & \mathsf{val}\ Empty \triangleright empty = Nil^{set}[] \\
 & & \mathsf{val}\ Singleton \triangleright singleton = \lambda x.Cons^{set}[x, Nil^{set}[]] \\
 & & \mathsf{val}\ Cmp \triangleright cmp = \lambda l_1.\lambda l_2. \\
 & & \quad \mathsf{match}^{set}_{Nil,Cons[elt,set]}\,[int]\, l_1 \\
 & & \qquad (\mathsf{match}^{set}_{Nil,Cons[elt,set]}\,[int]\, l_2 \\
 & & \qquad\ 0 \\
 & & \qquad\ \lambda hd.\lambda tl. -1) \\
 & & \qquad (\lambda hd_1.\lambda tl_1.\,\mathsf{match}^{set}_{Nil,Cons[elt,set]}\,[int]\, l_2 \\
 & & \qquad\ 1 \\
 & & \qquad\ \lambda hd_2.\lambda tl_2.\ \ (elt\_cmp \\
 & & \qquad\qquad\qquad\qquad (list.Max\, l_1) \\
 & & \qquad\qquad\qquad\qquad (list.Max\, l_2))) \\
 & & \ldots \quad \rangle
\end{array}
$$

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.

$$
\begin{array}{lll}
openSos = & \langle & \mathsf{type}\ Elt \triangleright elt : \star \\
 & & \mathsf{type}\ Set \triangleright set : \star \\
 & & \mathsf{type}\ Sos \triangleright sos : \star \\
 & & sos \prec Int[int], Set[set] \\
 & & \mathsf{val}\ Empty \triangleright empty : set \\
 & & \mathsf{val}\ Singleton \triangleright singleton : elt \to set \\
 & & \mathsf{val}\ Set\_cmp \triangleright set\_cmp : set \to set \to int \\
 & ; & \mathsf{type}\ Sos \triangleright sos' : \star \\
 & & sos' \prec Int[int], Set[set] \\
 & & \mathsf{type}\ Elt \triangleright elt' = sos \\
 & & \mathsf{val}\ Cmp \triangleright cmp = \lambda sos_1.\lambda sos_2.\text{[snipped code]} \quad \rangle
\end{array}
$$

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 it. Thus, *Sos* is obtained by

$$\begin{aligned} Sos = \mathsf{close}(\ & (openSet[Cmp \mapsto Set\_cmp] + \\ & openSos[Cmp \mapsto Elt\_cmp]) \\ & [Elt\_cmp \mapsto Cmp]) \end{aligned}$$

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.

```
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.

This program is written in a surface language, which is not the calculus Dreyer et al. studied. The program is therefore elaborated to this calculus, as we explain informally. SOS is elaborated into an opaque rds, roughly a rds that prohibits the use of recursive types. By phase-splitting, opaque rds's reduce to non-recursive signatures ; here SOS is roughly equivalent to

```
module type SOS′ = sig
  type sos
  type elt = sos
  type set = M { Key.key ↦ elt }
  val Int : int -> sos
  val Set : set -> sos
  val expose : sos -> (int + set)
end
```

which is not recursive. (Notice that Dreyer et al. use the opaque interpretation of datatypes.)

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 type definitions, possibly nested inside sub-modules. It is elaborated to an *opaque fixed-point*, which allows datatype definitions (see [29] for details). We obtain something like

```
module StaticSos = opaque struct rec Sos : SOS in
  type sos = Int of int | Set of Sos.SosSet.set
  module SosKey = struct
    type key = sos
  end
  module SosSet = struct
    type set = M { Key.key ↦ SosKey.key }
    ...
  end
end
```

The dynamic part of the module is then elaborated to a *transparent fixed-point*, which does not allow datatype definitions, since these are opaque, but is more flexible than the opaque fixed-point otherwise. A transparent fixed-point requires the signature of the recursive module variable (here Sos) to be fully transparent, so datatype definitions are elaborated by referring to their first elaboration in the StaticSos. We obtain

```
module Sos = transparent struct rec Sos : (SOS / StaticSos) in
  type sos = StaticSos.sos
  module SosKey = struct
    type key = StaticSos.sos
  end
  module SosSet = MkSet(SosKey)
end
```

Problem: in the source program, the type set in the result of the MkSet functor could be constrained to be abstract. In the proposed elaboration, it would then be impossible to extract the static 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 limitation comes from the choice they make to elaborate the dynamic part of the recursive module as a transparent fixed-point. This choice seems to be mainly guided by two facts.

- 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 (l : List.t) n =
      if n = 0 then l
      else match l 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 Dreyer et al. Here, one could implement the type of list without any module-recursive reference. However, in the case of datatype definitions split across different sub-modules, module-recursive references are needed. Thus, it is simpler to consider a single datatype 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 Dreyer 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-recursiveness 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]:

$$\frac{\begin{array}{c}\Gamma \vdash S \equiv [\alpha : \kappa.\sigma_1]\ \mathit{sig} \qquad \Gamma[s \uparrow S] \vdash M \equiv [c, e] \qquad \Gamma \vdash c \downarrow \kappa \\ \Gamma[s \uparrow [\alpha : \mathfrak{s}(c).\sigma_1]] \vdash e \downarrow \sigma_2 \qquad \Gamma[\alpha : \mathfrak{s}(c)] \vdash \sigma_1 \equiv \sigma_2[\alpha/(\mathit{Fst}\ s)]\ \mathit{type}\end{array}}{\Gamma \vdash \mathit{fix}_S(s : S)M : S}$$

(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 rds’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 MkSet returns an abstract type, because the underlying calculus used in [29] does not feature generativity. It would be useful to try and transpose the discussion to the more recent formalism of [28].

In *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 *MML* seems more natural than the one of [29]. In particular, bundles of recursive modules are dealt with in a very *ad hoc* way in [29], while they are encoded smoothly in *MML*.

5.4.4 Mathematical data structures

Presentation In [15, 14], in the context of the Foc project [1], 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 *entities*. In mathematics, entities are grouped in *collections*, which express a link between these entities, possibly materialized by operations called *methods*. For example, the entities $0, 1, 2, \ldots$ form the collection of natural numbers. Slighlty 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 *carrier*, or *representation type*. For natural numbers, it is int. Mathematical collections are in turn grouped by certain sets of properties, called *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 multiplicating polynomials, even if the carrier or the type of the coefficients are abstracted over. Species have *interfaces*, specifying the set of methods they define. For more details, see [15, 14, 62].

The aim of the Foc project is to develop a certified library by extraction of OCaml programs from Coq specifications. [?] (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 t, the carrier, and an object meth, containing the methods operating on t. When all methods of a species s are defined, it can be instantiated into a collection. For this, a module is created, which contains the corresponding carrier and the species s. For example, if the class 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 poly can be created by

```
module Poly = (struct
  type t = (int * float) list
  class meth = new s
end : sig
  type t
  class meth : st
end)
```

[1] http://www-spi.lip6.fr/~foc

where st is the type of s, abstracted over the carrier. This achieves abstraction over the representation of the carrier. Extensibility and refinement are allowed by operating on the class 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 ← 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 $I[(X \triangleright x) \mapsto (Y \triangleright y)]$ 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:

$$\begin{array}{l}\mathsf{type}\ \mathit{basic\_object\_sig} = \{\ \mathsf{type}\ T \triangleright t : \star, \\ \qquad\qquad\qquad\qquad\quad \mathsf{val}\ \mathit{Print} \triangleright \mathit{print} : t \to \mathit{unit}\ \ \}\end{array}$$

The basic species, at the top the semantic inheritance hierarchy of the structures we will define, is:

$$\begin{array}{lll}\mathsf{val}\ \mathit{basic\_object} = & \langle & \mathit{include}\ \mathit{basic\_object\_sig} \\ & ; & \epsilon\ \ \rangle\end{array}$$

The interface of a set is defined by the following module type:

$$\begin{array}{ll}\mathsf{type}\ \mathit{set\_sig} = \{ & \mathit{include}\ \mathit{basic\_object\_sig} \\ & \mathsf{val}\ \mathit{Eq} \triangleright \mathit{eq} : t \to t \to \mathit{bool} \\ & \mathsf{val}\ \mathit{Neq} \triangleright \mathit{neq} : t \to t \to \mathit{bool} \\ & \ \ \}\end{array}$$

The species of sets is the first to have a concrete method, *Neq*, which can be defined in terms of *Eq*:

$$\begin{array}{lll}\mathsf{val}\ \mathit{set} = \mathit{basic\_object} + & \langle & \mathit{include}\ \mathit{set\_sig} \\ & ; & \mathsf{val}\ \mathit{Neq} \triangleright \mathit{neq} = \lambda x.\lambda y.not(eq\,x\,y)\ \ \rangle\end{array}$$

We define the interface of partial orders as:

$$\begin{array}{ll}\mathsf{type}\ \mathit{partial\_order\_sig} = \{ & \mathit{include}\ \mathit{set\_sig} \\ & \mathsf{val}\ \mathit{Leq} \triangleright \mathit{leq} : t \to t \to \mathit{bool} \\ & \mathsf{val}\ \mathit{Lt} \triangleright \mathit{lt} : t \to t \to \mathit{bool} \\ & \mathsf{val}\ \mathit{Geq} \triangleright \mathit{geq} : t \to t \to \mathit{bool} \\ & \mathsf{val}\ \mathit{Gt} \triangleright \mathit{gt} : t \to t \to \mathit{bool}\ \ \}\end{array}$$

[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 imlement the three other ones. Thus, the species of partial orders can be defined as:

$$\begin{array}{lll}
\mathsf{val}\, \mathit{partial\_order} = \mathit{set} + & \langle & \mathit{include}\, \mathit{partial\_order\_sig} \\
 & ; & \mathsf{val}\, \mathit{Lt} \triangleright \mathit{lt} = \lambda x.\lambda y.(\mathit{leq}\, x\, y)\, \mathsf{and} (\mathit{not}(\mathit{eq}\, x\, y)) \\
 & & \mathsf{val}\, \mathit{Geq} \triangleright \mathit{geq} = \lambda x.\lambda y.(\mathit{leq}\, y\, x) \\
 & & \mathsf{val}\, \mathit{Gt} \triangleright \mathit{gt} = \lambda x.\lambda y.(\mathit{lt}\, y\, x) \quad \rangle
\end{array}$$

Lattices must match the same interface as partial orders, with two additional functions, the greatest lower bound and the least upper bound functions:

$$\begin{array}{ll}
\mathsf{type}\, \mathit{lattice\_sig} = \{ & \mathit{include}\, \mathit{partial\_order\_sig} \\
 & \mathsf{val}\, \mathit{Glb} \triangleright \mathit{glb} : t \to t \to t \\
 & \mathsf{val}\, \mathit{Lub} \triangleright \mathit{lub} : t \to t \to t \quad \}
\end{array}$$

The species of lattices does not have anything to define by default, and is therefore implemented as:

$$\begin{array}{lll}
\mathsf{val}\, \mathit{lattice} = \mathit{partial\_order} + & \langle & \mathit{include}\, \mathit{lattice\_sig} \\
 & ; & \epsilon \quad \rangle
\end{array}$$

Then, the interfaces for mix- and max-lattices add the distinguished elements *Min* and *Max*, respectively:

$$\begin{array}{ll}
\mathsf{type}\, \mathit{min\_lattice\_sig} = \{ & \mathit{include}\, \mathit{lattice\_sig} \\
 & \mathsf{val}\, \mathit{Min} \triangleright \mathit{min} : t \\
 & \mathsf{val}\, \mathit{Is\_min} \triangleright \mathit{is\_min} : t \to \mathit{bool} \quad \} \\
\mathsf{type}\, \mathit{max\_lattice\_sig} = \{ & \mathit{include}\, \mathit{lattice\_sig} \\
 & \mathsf{val}\, \mathit{Max} \triangleright \mathit{max} : t \\
 & \mathsf{val}\, \mathit{Is\_max} \triangleright \mathit{is\_max} : t \to \mathit{bool} \quad \}
\end{array}$$

The corresponding species can define the methods *Is_min* and *Is_max*, respectively, in terms of *Mix* and *Max*:

$$\begin{array}{lll}
\mathsf{val}\, \mathit{min\_lattice} = \mathit{lattice} + & \langle & \mathit{include}\, \mathit{min\_lattice\_sig} \\
 & ; & \mathsf{val}\, \mathit{Is\_min} \triangleright \mathit{is\_min} = \lambda x.(\mathit{eq}\, x\, \mathit{min}) \quad \rangle \\
\mathsf{val}\, \mathit{max\_lattice} = \mathit{lattice} + & \langle & \mathit{include}\, \mathit{max\_lattice\_sig} \\
 & ; & \mathsf{val}\, \mathit{Is\_max} \triangleright \mathit{is\_max} = \lambda x.(\mathit{eq}\, x\, \mathit{max}) \quad \rangle
\end{array}$$

Complete lattices can be implemented by inheriting both from min- and max-lattices.

$$\begin{array}{ll}
\mathsf{type}\, \mathit{complete\_lattice\_sig} = \{ & \mathit{include}\, \mathit{max\_lattice\_sig} \\
 & \mathit{include}\, \mathit{min\_lattice\_sig} \quad \} \\
\mathsf{val}\, \mathit{complete\_lattice} = \mathit{max\_lattice} \leftarrow \mathit{min\_lattice} &
\end{array}$$

The species of complete lattices has the mixin module type

$$
\begin{array}{llll}
\langle & \mathsf{type}\, T \triangleright t : \star & ;\ \mathsf{val}\, Neq \triangleright neq : t \to t \to bool & ;\to;\emptyset\rangle \\
& \mathsf{val}\, Print \triangleright print : t \to unit & \ \ \mathsf{val}\, Lt \triangleright lt : t \to t \to bool & \\
& \mathsf{val}\, Eq \triangleright eq : t \to t \to bool & \ \ \mathsf{val}\, Geq \triangleright geq : t \to t \to bool & \\
& \mathsf{val}\, Neq \triangleright neq : t \to t \to bool & \ \ \mathsf{val}\, Gt \triangleright gt : t \to t \to bool & \\
& \mathsf{val}\, Leq \triangleright leq : t \to t \to bool & \ \ \mathsf{val}\, Is\_min \triangleright is\_min : t \to bool & \\
& \mathsf{val}\, Lt \triangleright lt : t \to t \to bool & \ \ \mathsf{val}\, Is\_max \triangleright is\_max : t \to bool & \\
& \mathsf{val}\, Geq \triangleright geq : t \to t \to bool & & \\
& \mathsf{val}\, Gt \triangleright gt : t \to t \to bool & & \\
& \mathsf{val}\, Glb \triangleright glb : t \to t \to t & & \\
& \mathsf{val}\, Lub \triangleright lub : t \to t \to t & & \\
& \mathsf{val}\, Min \triangleright min : t & & \\
& \mathsf{val}\, Is\_min \triangleright is\_min : t \to bool & & \\
& \mathsf{val}\, Max \triangleright max : t & & \\
& \mathsf{val}\, Is\_max \triangleright is\_max : t \to bool & &
\end{array}
$$

(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:

$$
\begin{array}{lll}
\mathsf{val}\, open\_int\_lattice = complete\_lattice + & \langle & \emptyset \\
& ; & \mathsf{type}\, T \triangleright t = int \\
& & \mathsf{val}\, Print \triangleright print = print_i nt \\
& & \mathsf{val}\, Eq \triangleright eq = \lambda x.\lambda y.(x = y) \\
& & \mathsf{val}\, Leq \triangleright leq = \lambda x.\lambda y.(x \leq y) \\
& & \mathsf{val}\, Glb \triangleright glb = \lambda x.\lambda y.\mathsf{if}\ \ x \leq y\ \ \mathsf{then}\ \ x\ \ \mathsf{else}\ \ y \\
& & \mathsf{val}\, Lub \triangleright lub = \lambda x.\lambda y.\mathsf{if}\ \ x \geq y\ \ \mathsf{then}\ \ x\ \ \mathsf{else}\ \ y \\
& & \mathsf{val}\, Min \triangleright min = 0 \\
& & \mathsf{val}\, Max \triangleright max = 10 \quad \rangle \\
\mathsf{val}\, int\_lattice = \mathsf{close}\, open\_int\_lattice & &
\end{array}
$$

We can then decide that the algorithm for Lt is too inefficient, and incrementally implement an optimized collection *optimized_int_lattice*, with the comparison function from the library, as follows.

$$
\begin{array}{lll}
\mathsf{val}\, open\_optimized\_int\_lattice = open\_int\_lattice \leftarrow & \langle & include\, partial\_order\_sig \\
& ; & \mathsf{val}\, Lt \triangleright lt = \lambda x.\lambda y.(x < y) \quad \rangle \\
\mathsf{val}\, optimized\_int\_lattice = \mathsf{close}\, open\_optimized\_int\_lattice & &
\end{array}
$$

Hard example (part VII): recursive polynomials, a first attempt A very subtle example of a representation of mathematical structures is given in [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:

$$\begin{array}{ll}
\mathsf{type}\, ring\_sig = \{ & \mathsf{type}\, T \triangleright t \\
& \mathsf{val}\, Eq \triangleright eq : t \to t \to bool \\
& \mathsf{val}\, Zero \triangleright zero : t \\
& \mathsf{val}\, Eq\_zero \triangleright eq\_zero : t \to bool \\
& \mathsf{val}\, Un \triangleright un : t \\
& \mathsf{val}\, Add \triangleright add : t \to t \to t \\
& \mathsf{val}\, Minus \triangleright minus : t \to t \to t \\
& \mathsf{val}\, Uminus \triangleright uminus : t \to t \\
& \mathsf{val}\, Mult \triangleright mult : t \to t \to t \quad \}
\end{array}$$

The natural module type for polynomials has the ring of its coefficients as a virtual component, and some more functionalities related to polynomials:

$$\begin{array}{ll}
\mathsf{type}\, poly\_sig = \{ & \mathsf{val}\, Coef \triangleright coef : ring\_sig \\
& include\, ring\_sig \\
& \mathsf{val}\, Lift \triangleright lift : coef.T \to t \\
& \mathsf{val}\, Mult\_extern \triangleright mult\_extern : coef.T \to t \to t \\
& \mathsf{val}\, Lc \triangleright lc : t \to coef.T \\
& \mathsf{val}\, Is\_coef \triangleright is\_coef : t \to bool \quad \}
\end{array}$$

The $lift$ function lifts a coefficient to a polynomial of degree zero. The $mult\_extern$ 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:

$$\begin{array}{lll}
\mathsf{val}\, poly = & \langle & include\, poly\_sig \\
& ; & \mathsf{val}\, Mult\_extern \triangleright mult\_extern = \lambda c.\lambda p.(mult\,(lift\, c)\, p) \\
& & \mathsf{val}\, Eq\_zero \triangleright eq\_zero = \lambda p.(coef.Eq\_zero\,(lc\, p)) \\
& & \mathsf{val}\, Is\_coef \triangleright is\_coef = \lambda p.(eq\, p\,(lift\,(lc\, p))) \quad \rangle
\end{array}$$

We can now define the mixin module of recursive polynomials. It relies on a representation of polynomials $Poly \triangleright my\_poly$ (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 $Base \triangleright base$ 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 $Rec\_poly \triangleright rec\_poly$, defining the coefficients of the import module my_poly, i.e. the recursive polynomials. This is specified by the type sharing equation $\mathsf{with\,type}\; Coef.T = support$ 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 $(Coef \triangleright coef)$ has been renamed to $(Base \triangleright my\_base)$, in order both to match the fact that it re-exports the imported module $Base \triangleright base$, and to avoid conflict with its internal variable $base$. It specializes the type T of $poly$ to $support$. The intersting functions are $Compose$ and Add.

```
val $poly\_rec =$ $\langle$ val $Base \triangleright base : ring\_sig$
                   val $Poly \triangleright my\_poly : poly\_sig$ with type $Coef.T = support$
                   type $Support \triangleright support : \star$
                   $support \prec \Phi$
                 ; type $Support \triangleright support' = \star$
                   $support' \prec \Phi$
                   val $Rec\_poly \triangleright rec\_poly =$ close(
                     $poly[(Coef \triangleright coef) \mapsto (Base \triangleright my\_base)] \leftarrow$
                     $\langle$ $include\, poly\_sig[(Coef \triangleright coef) \mapsto (Base \triangleright my\_base)]$
                           with type    $T = support$
                            and type    $Base.T = base.T$
                       val $Compose \triangleright compose : string \to my\_poly.T \to t$
                     ; val $Base \triangleright my\_base' = base$
                       type $T \triangleright t' = support$
                       val $Eq \triangleright eq' = \lambda x.\lambda y.(x = y)$
                       val $Zero \triangleright zero' = Base^{t}[base.Zero]$
                       val $Un \triangleright un' = Base^{t}[base.Un]$
                       val $Compose \triangleright compose' = \lambda v.\lambda l.$
                         if  $my\_poly.Is\_coef\; l$  then  $my\_poly.Lc\; l$
                         else  $Comp^{t}[v, l]$
                       val $Lift \triangleright lift' = \lambda a.Base^{t}[a]$
                       val $Lc \triangleright lc' = \lambda x.$ match$^{t}_{\Phi}$ $[base.T]\; x$
                         $\lambda a.a$
                         $\lambda v.\lambda l.(lc\; (my\_poly.Lc\; l))$
                       val $Add \triangleright add' = \lambda x_1.\lambda x_2.$ match$^{t}_{\Phi}$ $[t]\; x_1$
                         $\lambda a_1.$ match$^{t}_{\Phi}$ $[t]\; x_2$
                               $\lambda a_2.Base^{base.T}[base.Add\; a_1\; a_2]$
                               $\lambda v_2.\lambda l_2.(compose\; v_2\; (my\_poly.Add\; (my\_poly.Lift\; x_1)\; l_2))$
                         $\lambda v_1.\lambda l_1.$ match$^{t}_{\Phi}$ $[t]\; x_2$
                          $\lambda a_2.(compose\; v_1\; (my\_poly.Add\; (my\_poly.Lift\; x_2)\; l_1))$
                          $\lambda v_2.\lambda l_2.$ if  $v_1 = v_2$  then  $(compose\; v_1\; (my\_poly.Add\; l_1\; l_2))$
                             else  if  $v_1 > v_2$
                             then  $(compose\; v_1\; (my\_poly.Add\; (my\_poly.Lift\; x_2)\; l_1))$
                             else  $(compose\; v_2\; (my\_poly.Add\; (my\_poly.Lift\; x_1)\; l_2))$
                     [ ...snipped ... ]  $\rangle$  )  $\rangle$
```

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 $Comp^t[v, l]$.

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.
- It both arguments are composed polynomials, i.e. constructed with the *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 *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 *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 *compose* function.

Until now, no problem arose. But assume now that we have implemented the ring of integers *int_ring* and a mixin module for sparse polynomials *sparse_poly*. If we try to construct recursive polynomials by composing these two mixin modules with *poly_rec*, we write

$$
\begin{array}{lll}
\mathsf{val}\ \mathit{try} = & \mathit{poly\_rec} + & \\
& \langle & \mathsf{val}\ \mathit{Rec\_poly} \triangleright \mathit{rec\_poly} : \mathit{poly\_sig} \\
& ; & \mathsf{val}\ \mathit{Poly} \triangleright \mathit{my\_poly} = \mathsf{close}(\ \mathit{sparse\_poly} + \\
& & \qquad\qquad\qquad\qquad \langle \emptyset ; \mathsf{val}\ \mathit{Coef} \triangleright \mathit{coef} = \mathit{rec\_poly} \rangle) \\
& & \mathsf{val}\ \mathit{Base} \triangleright \mathit{base} = \mathit{int\_ring} \quad \rangle
\end{array}
$$

Unfortunately, this expression is ill-typed, since there is a dependency cycle between *rec_poly* and *my_poly*, and both are expressions of the shape close..., 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 *sparse_poly*? They perfectly could require some components of *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 *ring_sig*, *poly_sig* module types, and the *poly* and mixin module are defined as above, except that the sub-module representing coefficients is now inlined in *poly_sig* (and consequently also in *poly*). The modified module type is

$$\begin{array}{ll}\mathsf{type}\; \mathit{poly\_sig} = \{ & \mathit{include}\, \mathit{ring\_sig}[(\mathit{Coef\_} \triangleright \mathit{coef\_}) \cdot (X \triangleright x)] \\ & \mathit{include}\, \mathit{ring\_sig} \\ & \mathsf{val}\; \mathit{Lift} \triangleright \mathit{lift} : \mathit{coef} \rightarrow t \\ & \mathsf{val}\; \mathit{Mult\_Extern} \triangleright \mathit{mult\_extern} : \mathit{coef} \rightarrow t \rightarrow t \\ & \mathsf{val}\; \mathit{Lc} \triangleright \mathit{lc} : t \rightarrow \mathit{coef} \\ & \mathsf{val}\; \mathit{Is\_Coef} \triangleright \mathit{is\_coef} : t \rightarrow \mathit{bool} \quad \} \end{array}$$

Coefficients are represented by the included signature $\mathit{ring\_sig}[(\mathit{Coef\_} \triangleright \mathit{coef\_}) \cdot (X \triangleright x)]$, which brings the type $\mathit{Coef\_T}$ of coefficients, and ring operations on it, such as $\mathit{Coef\_Mult}$ and $\mathit{Coef\_Add}$. 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 $\mathit{ring\_sig}$ signature, prefixed with $\mathit{Base\_}$, to mimick the imported Base sub-module of the first attempt. Similarly, the sub-module Poly of the first attempt is imported here as a $\mathit{poly\_sig}$ signature. The type sharing equation $\mathit{Coef}.T = \mathit{support}$ is converted into a renaming removing the prefix of all the components beginning with $\mathit{Poly\_Coef\_}$: this makes them match the comonents corresponding to recursive polynomials. The main datatype is then defined, but must be modified according to the new naming conventions: $\Phi' = \mathit{Base}[\mathit{base\_t}], \mathit{Comp}[\mathit{string}, \mathit{poly\_t}]$. 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 $\mathit{int\_ring}$ and a mixin module for sparse polynomials $\mathit{sparse\_poly}$ (which has been flattened to match the signature $\mathit{poly\_sig}$). The code is as follows:

$$\begin{array}{l} \mathsf{val}\; \mathit{int\_recursive\_polynomials} = (\mathit{int\_ring}[(\mathit{Base\_} \triangleright \mathit{base\_}) \cdot (X \triangleright x)] \\ \;+ (\mathit{sparse\_poly} \;\; [(\mathit{Poly\_} \triangleright \mathit{poly\_}) \cdot (X \triangleright x)] \\ \qquad\qquad\qquad \begin{array}{lrclrcl} [\; \mathsf{type} & \mathit{Poly\_Coef\_T} & \mapsto & T & & & \\ \;\; \mathsf{val} & \mathit{Poly\_Coef\_Eq} & \mapsto & \mathit{Eq} & \mathit{Poly\_Coef\_Zero} & \mapsto & \mathit{Zero} \\ & \mathit{Poly\_Coef\_Eq\_zero} & \mapsto & \mathit{Eq\_zero} & \mathit{Poly\_Coef\_Un} & \mapsto & \mathit{Un} \\ & \mathit{Poly\_Coef\_Add} & \mapsto & \mathit{Add} & \mathit{Poly\_Coef\_Minus} & \mapsto & \mathit{Minus} \\ & \mathit{Poly\_Coef\_Uminus} & \mapsto & \mathit{Uminus} & \mathit{Poly\_Coef\_Mult} & \mapsto & \mathit{Mult}\;]) \end{array} \\ \;+ \mathit{rec\_poly\_flat});; \end{array}$$

```
val $rec\_poly\_flat =$
 $poly[$ type          $Coef\_T \mapsto Base\_T$
         val          $Coef\_Eq \mapsto Base\_Eq$              $Coef\_Zero \mapsto Base\_Zero$
               $Coef\_Eq\_zero \mapsto Base\_Eq\_zero$      $Coef\_Un \mapsto Base\_Un$
                   $Coef\_Add \mapsto Base\_Add$           $Coef\_Minus \mapsto Base\_Minus$
               $Coef\_Uminus \mapsto Base\_Uminus$       $Coef\_Mult \mapsto Base\_Mult\ ]$
 $\leftarrow$
 $\langle$ $include\, ring\_sig[(Base\_ \triangleright base\_) \cdot (X \triangleright x)]$
   $include(poly\_sig$ with type $Coef\_T = support$
        $[(Poly\_ \triangleright poly\_) \cdot (X \triangleright x)]$
        $[$ type          $Poly\_Coef\_T \mapsto T$
           val          $Poly\_Coef\_Eq \mapsto Eq$              $Poly\_Coef\_Zero \mapsto Zero$
                 $Poly\_Coef\_Eq\_zero \mapsto Eq\_zero$      $Poly\_Coef\_Un \mapsto Un$
                     $Poly\_Coef\_Add \mapsto Add$           $Poly\_Coef\_Minus \mapsto Minus$
                 $Poly\_Coef\_Uminus \mapsto Uminus$       $Poly\_Coef\_Mult \mapsto Mult\ ])$
   val $Lift \triangleright lift : base\_t \to t$
   val $Mult\_Extern \triangleright mult\_extern : base\_t \to t \to t$
   val $Lc \triangleright lc : t \to base\_t$
   val $Is\_Coef \triangleright is\_coef : t \to bool$
   val $Compose \triangleright compose : string \to poly\_t \to t$
   type $Support \triangleright support : \star$
   $support \prec \Phi'$
 ; type $Support \triangleright support' : \star$
   $support' \prec \Phi'$ type $T \triangleright t' = support$
   val $Eq \triangleright eq' = \lambda x.\lambda y.(x = y)$
   val $Zero \triangleright zero' = Base^{t}[base\_zero]$
   val $Un \triangleright un' = Base^{t}[base\_un]$
   val $Compose \triangleright compose' = \lambda v.\lambda l.$
       if $poly\_is\_coef\ l$ then $poly\_lc\ l$
       else $Comp^{t}[v, l]$
   val $Lift \triangleright lift' = \lambda a.Base^{t}[a]$
   val $Lc \triangleright lc' = \lambda x.$ $\mathsf{match}^{t}_{\Phi}$ $[base\_t]\, x$
      $\lambda a.a$
      $\lambda v.\lambda l.(lc\ (poly\_lc\ l))$
   val $Add \triangleright add' = \lambda x_1.\lambda x_2.$ $\mathsf{match}^{t}_{\Phi}$ $[t]\, x_1$
      $\lambda a_1.$ $\mathsf{match}^{t}_{\Phi}$ $[t]\, x_2$
            $\lambda a_2.Base^{base\_t}[base\_add\ a_1\ a_2]$
            $\lambda v_2.\lambda l_2.(compose\ v_2\ (poly\_add\ (poly\_lift\ x_1)\ l_2))$
      $\lambda v_1.\lambda l_1.$ $\mathsf{match}^{t}_{\Phi}$ $[t]\, x_2$
       $\lambda a_2.(compose\ v_1\ (poly\_add\ (poly\_lift\ x_2)\ l_1))$
       $\lambda v_2.\lambda l_2.$ if $v_1 = v_2$ then $(compose\ v_1\ (poly\_add\ l_1\ l_2))$
          else if $v_1 > v_2$
          then $(compose\ v_1\ (poly\_add\ (poly\_lift\ x_2)\ l_1))$
          else $(compose\ v_2\ (poly\_add\ (poly\_lift\ x_1)\ l_2))$
   [ ...snipped ... ] $\rangle$   $\rangle$
```

Figure 5.16: Flattened recursive polynomials

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:

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

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

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

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

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

Notice that the let rec definition we generate is unusual: it involves function applications in the right-hand sides, which is usually not supported in call-by-value λ-calculi.

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 $\lambda_\circ$-calculus, featuring a let rec construct that slightly extends that of *MM*. It allows to group all the components within a single binding:

$$
\begin{array}{lll}
 & x \ \in \mathit{Vars} & \text{Variable} \\
 & X \ \in \mathit{Names} & \text{Name} \\
\text{Expression:} & e ::= x & \text{Variable} \\
 & \quad \mid \{X_1 = e_1 \ldots X_n = e_n\} & \text{Record} \\
 & \quad \mid e.X & \text{Record selection} \\
 & \quad \mid \mathsf{let\ rec}\ x_1 = e_1 \ldots x_n = e_n \ \mathsf{in}\ e & \mathsf{let\ rec} \\
 & \quad \mid \langle X_1 \triangleright x_1 \ldots X_n \triangleright x_n ; d_1 \ldots d_m \rangle & \text{Structure} \\
 & \quad \mid e_1 + e_2 \mid \mathsf{close}\, e & \text{Composition, closure} \\
 & \quad \mid e_{|X_1 \ldots X_n} \mid e_{|-X_1 \ldots X_n} & \text{Projection, deletion} \\
 & \quad \mid e[X_1 \mapsto Y_1 \ldots X_n \mapsto Y_n] & \text{Renaming} \\
 & \quad \mid e_{X \succ Y} & \text{Splitting} \\
 & & \\
\text{Definition:} & d ::= X[x_1 \ldots x_n] \triangleright x = e & \text{Named definition}
\end{array}
$$

Figure 6.1: Syntax of MM_e

```
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 $\lambda_\circ$, and show that the expressions generated by our compilation scheme are well-typed. However, the type system of *MM* would not accept them, so we have to find a finer type system. Fortunately, Boudol [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 $\lambda_\circ$, 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: MM_e

The syntax of MM_e terms and types is defined in figure 6.1. The meaning of meta-variables is kept from the presentation of *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 *MM*. The notion of syntactic correctness is maintained identical as for *MM*, and expressions are similarly identified modulo correct variable renaming.

The operational semantics are defined exactly as for *MM*, without the contraction rules Freeze, Hide, Show, and letting the meta-variable *op* range over the restricted set of operators (see figure 3.2), and denote by $op[e]$ the application of *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 *Degree* function returns ☹ on all kinds of expressions, except on mixin modules and records, where it returns ☺.

The definition of the type system slightly differs from that of *MM*. Indeed, the output sections of mixin module types are now lists of types, indexed by names, as indicated in figure 6.2. They

$$\begin{array}{lll} M \in \mathit{Types} & ::= & \{O\} \mid \langle I; O; G\rangle \\ I & \in & \mathit{Names} \xrightarrow{\text{Fin}} \mathit{Types} \\ O & ::= & \epsilon \mid X \mapsto M, O \\ G & \subset_{\text{Fin}} & \{X \xrightarrow{\chi} Y \mid X, Y \in \mathit{Names}, \chi \in \mathit{Degrees}\} \\ \Gamma & \in & \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{Types} \end{array}$$

Figure 6.2: Types for MM_{e}

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

$$\frac{(X_i, x_i) \in \mathit{dom}(\langle \iota; o\rangle) \qquad (X[x_1 \ldots x_n] \triangleright x = e) \in o}{X_i \overset{\frownie}{\longrightarrow}_{\langle \iota; o\rangle} X}$$

Figure 6.3: Dependencies in a 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 $\langle \iota; o\rangle$, 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 $\rightarrow_{\langle \iota; o\rangle}$ of a structure $\langle \iota; o\rangle$ 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 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 Γ are well-formed, and $\Gamma \vdash e : M$, then M is well-formed.*

6.2.2 The target language $\lambda_\circ$

The target language for our translation is the $\lambda_\circ$ calculus, a variant of the λ-calculus with records and recursive definitions introduced by Boudol [13].

Syntax

The syntax of $\lambda_\circ$ is defined in figure 6.4. Intuitively, it is a subset of 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

$$
\begin{array}{rcll}
x & \in & \mathit{Vars} & \text{Variable} \\
X & \in & \mathit{Names} & \text{Name} \\
\diamond & := & =_{[n]} \mid =_{[?]} & (n \text{ a natural}) \\
\multicolumn{4}{l}{\text{Expression:}} \\
e \in \mathit{expr} & ::= & x \mid \lambda x.e \mid e_1 e_2 & \\
 & \mid & \{X_1 = e_1 \ldots X_n = e_n\} & \\
 & \mid & e.X & \\
 & \mid & \mathsf{let\ rec}\, x_1 i s_1 e_1 \ldots x_n \diamond_n e_n & \\
 & & \mathsf{in}\, e &
\end{array}
$$

Figure 6.4: Syntax of $\lambda_\circ$

- More meta-variables:

$$
\begin{array}{rcll}
s & ::= & X_1 = e_1 \ldots X_n = e_n & \text{Record} \\
b & ::= & x_1 \diamond_1 e_1 \ldots x_n \diamond_n e_n & \text{Binding}
\end{array}
$$

- Notations:

For a finite map f, and a set of variables P,

$\mathit{dom}(f)$ is its domain,
$\mathit{cod}(f)$ is its codomain
$f_{|P}$ is its restriction to P,
and $f_{\backslash P}$ is its restriction to $\mathit{dom}(f) \setminus P$.

- Expressions of predictable shape:

$e_{\downarrow} \in \mathit{Predictable} ::= \{o\} \mid \lambda x.e \mid \mathsf{let\ rec}\ b\ \mathsf{in}\ e_{\downarrow}$

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 $\{X_1 = e_1 \dots X_n = e_n\}$, record selection $e.X$ and a $\mathsf{let\ rec}$ construct. A mutually recursive definition has the shape $\mathsf{let\ rec}\ x_1 \diamond_1 e_1 \dots x_n \diamond_n e_n\ \mathsf{in}\ e$, where arbitrary expressions are syntactically allowed as the right-hand side of a definition.

Syntactic correctness Records $s = (X_1 = e_1 \dots X_n = e_n)$ and bindings $b = (x_1 \diamond_1 e_1 \dots x_n \diamond_n e_n)$ 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.

In a $\mathsf{let\ rec}$ binding $b = (x_1 = e_1 \dots x_n = e_n)$, we say that there is a forward reference from x_i to x_j if $1 \leq i \leq j \leq n$ and $x_j \in FV(e_i)$. 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 $e_{\downarrow} \in \mathit{Predictable} ::= \{s\} \mid \lambda x.e \mid \mathsf{let\ rec}\ b\ \mathsf{in}\ e_{\downarrow}$.

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 *dom*, the codomain *cod*, the restriction $\cdot_{|P}$ to a set P, or the co-restriction $\cdot_{\backslash P}$ outside of a set P.

Structural equivalence We consider the expressions equivalent up to alpha-conversion of binding variables in structures and $\mathsf{let\ rec}$ expressions. The set of terms of $\lambda_\circ$ is defined as the set of structural equivalence classes.

Semantics

The semantics of $\lambda_\circ$ is quite similar to that of MM_{e}, except for what concerns $\mathsf{let\ rec}$ 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 $\lambda_\circ$ denotes such fully evaluated bindings. This implicitely 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 $X_1 = v_1 \dots X_n = v_n$.

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* $\leadsto_c$. 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 $\mathsf{let\ rec}$.

In $\lambda_\circ$, for mostly technical reasons, we distinguish the topmost binding syntactically : the global computational reduction relation $\dashrightarrow_c$ 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 $\mathsf{let\ rec}$ and the notion of evaluation contexts are adapted to this notion of configuration. The computational contraction rule LIFT remains the same. The computational

Configuration:

$$c \quad ::= \quad b \vdash e$$

Value:

$$v \in \mathit{values} \quad ::= \quad x \mid \lambda x.e \mid \{s_v\}$$

Answer:

$$a \in \mathit{answers} \quad ::= \quad b_v \vdash v$$

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 6.6: Configurations and answers in $\lambda_\circ$

- **Computational contraction rules**

$$\{X_1 = v_1 \ldots X_n = v_n\}.X_i \leadsto_c v_i \quad (\textsc{Project}) \qquad \frac{x \notin FV(v)}{(\lambda x.e)v \leadsto_c \textsf{let rec } x = v \textsf{ in } e} \quad (\textsc{Beta})$$

$$\frac{dom(b) \perp FV(\mathbb{L})}{\mathbb{L}[\textsf{let rec } b \textsf{ in } e] \leadsto_c \textsf{let rec } b \textsf{ in } \mathbb{L}[e]} \quad (\textsc{Lift})$$

- **Computational reduction rules**

$$\frac{e \leadsto_c e'}{\mathbb{E}[e] \dashrightarrow_c \mathbb{E}[e']} \quad (\textsc{Context})$$

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

$$\frac{dom(b) \perp (dom(b_v) \cup FV(b_v))}{(b_v \vdash \textsf{let rec } b \textsf{ in } e) \dashrightarrow_c b_v, b \vdash e} \quad (\text{EM}) \qquad \frac{\mathbb{E}[\mathbb{N}](x) = v}{\mathbb{E}[\mathbb{N}[x]] \dashrightarrow_c \mathbb{E}[\mathbb{N}[v]]} \quad (\textsc{Subst})$$

- **Evaluation contexts**

Lift context:

$$\mathbb{L} \quad ::= \quad \Box e \mid v\Box \mid \Box.X \mid \{\mathbb{S}\}$$

Nested lift context:

$$\mathbb{F} \quad ::= \quad \Box \mid \mathbb{L}[\mathbb{F}]$$

Evaluation context:

$$\mathbb{E} \quad ::= \quad (b_v \vdash \mathbb{F}) \mid (\mathbb{B}[\mathbb{F}] \vdash e)$$

Record contexts:

$$\mathbb{S} \quad ::= \quad s_v, X = \Box, s$$

Binding contexts:

$$\mathbb{B} \quad ::= \quad b_v, x = \Box, b$$

Strict contexts:

$$\mathbb{N} \quad ::= \quad \Box v \mid \Box.X$$

- **Access in evaluation contexts**

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

Figure 6.7: Reduction semantics for $\lambda_\circ$

reduction relation extends the computational contraction relation to any evaluation context $\mathbb{E}$, as defined in figure 6.7. An evaluation context $\mathbb{E}$ 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 $\longrightarrow$.

Definition 18 *The $\lambda_\circ$ language is the set of terms, equipped with the relation $\longrightarrow$.*

$\lambda_\circ$ 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 λ-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 MM_e terms into call-by-value λ-calculus extended with records and a let rec binding. This compilation scheme is compositional, and type-directed, thus supporting separate compilation.

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 $\langle\iota; o\rangle$ is translated into a record construction. The resulting record has one field for each exported name $X \in dom(o)$, and this field is associated to $o(X)$ where all input parameters on which X depends are λ-abstracted. Some notation is required here. We write $D^{-1}(X)$ 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 $\iota^{-1}(D^{-1}(X))$ for the list $(x_1, \ldots, x_n)$ of variables associated to the names $(X_1, \ldots, X_n)$ by the input mapping ι. Finally, we write $\vec{\lambda}(x_1, \ldots, x_n).M$ as shorthand for $\lambda x_1 \ldots \lambda x_n.M$. With all this notation, the field X in the record translating $\langle\iota; o\rangle$ is bound to $\vec{\lambda}\iota^{-1}(D^{-1}(X)).[\![o(X) : O(X)]\!]$.

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 $M\ x_1\ \ldots\ x_n$.

The freeze operation $E\,!\,X$ is perhaps the hardest to compile. Output components Z that do not depend on X are simply re-exported from $[\![E]\!]$. For the other output components, consider a component Y of E that depends on $Y_1, \ldots, Y_n$, and assume that one of these dependencies is X, which itself depends on $X_1, \ldots, X_p$. In $E\,!\,X$, the Y component depends on $(\{Y_i\} \cup \{X_j\}) \setminus \{X\}$.

$$
\begin{array}{l}
\llbracket (e : M').X : M \rrbracket = \llbracket e : M' \rrbracket .X \\
\llbracket \langle \iota ; o \rangle : \{I; O; D\} \rrbracket = \\
\qquad \{X = \vec{\lambda}\iota^{-1}(D^{-1}(X)).\llbracket o(X) : O(X) \rrbracket \mid X \in dom(O)\} \\
\llbracket (E_1 : \{I_1; O_1; D_1\}) + (E_2 : \{I_2; O_2; D_2\}) : \{I; O; D\} \rrbracket = \\
\qquad \mathsf{let}\ e_1 = \llbracket E_1 : \{I_1; O_1; D_1\} \rrbracket\ \mathsf{in}\ \mathsf{let}\ e_2 = \llbracket E_2 : \{I_2; O_2; D_2\} \rrbracket\ \mathsf{in} \\
\qquad \langle X = e_1.X \mid X \in dom(O_1); \\
\qquad\ \ Y = e_2.Y \mid Y \in dom(O_2)\rangle \\
\llbracket (E : \{I'; O'; D'\}) \setminus X : \{I; O; D\} \rrbracket = \\
\qquad \mathsf{let}\ e = \llbracket E : \{I'; O'; D'\} \rrbracket\ \mathsf{in}\ \langle Y = e.Y \mid Y \in dom(O)\rangle \\
\llbracket (E : \{I'; O'; D'\})[X \leftarrow Y] : \{I; O; D\} \rrbracket = \\
\qquad \mathsf{let}\ e = \llbracket E : \{I'; O'; D'\} \rrbracket\ \mathsf{in} \\
\qquad \langle Z\{X\}Y = \vec{\lambda}\overline{D^{-1}(Z\{X\}Y)}.(e.Z\ \overline{D'^{-1}(Z)})\{\overline{X}\}\overline{Y} \mid Z \in dom(O')\rangle \\
\llbracket (E : \{I'; O'; D'\})\,!\,X : \{I; O; D\} \rrbracket = \\
\qquad \mathsf{let}\ e = \llbracket E : \{I'; O'; D'\} \rrbracket\ \mathsf{in} \\
\qquad \langle Z = e.Z \mid Z \in dom(O),\ X \notin D'^{-1}(Z); \\
\qquad\ \ Y = \vec{\lambda}\overline{D^{-1}(Y)}.\mathsf{let\ rec}\ \overline{X} = e.X\ \overline{D'^{-1}(X)}\ \mathsf{in}\ e.Y\ \overline{D'^{-1}(Y)} \mid X \in D'^{-1}(Y)\rangle \\
\llbracket \mathsf{close}\, E : \{I'; O'; D'\} : \{\emptyset; O; \emptyset\} \rrbracket = \\
\qquad \mathsf{let}\ e = \llbracket E : \{I'; O'; D'\} \rrbracket\ \mathsf{in} \\
\qquad \mathsf{let\ rec}\ \overline{X^1_1} = e.X^1_1\ \overline{D'^{-1}(X^1_1)}\ \mathsf{and}\ \ldots\ \mathsf{and}\ \overline{X^1_{n_1}} = e.X^1_{n_1}\ \overline{D'^{-1}(X^1_{n_1})}\ \mathsf{in} \\
\qquad \ldots \\
\qquad \mathsf{let\ rec}\ \overline{X^p_1} = e.X^p_1\ \overline{D'^{-1}(X^p_1)}\ \mathsf{and}\ \ldots\ \mathsf{and}\ \overline{X^p_{n_p}} = e.X^p_{n_p}\ \overline{D'^{-1}(X^p_{n_p})}\ \mathsf{in} \\
\qquad \langle X = \overline{X} \mid X \in dom(O)\rangle \\
\qquad \text{where } (\{X^1_1 \ldots X^1_{n_1}\}, \ldots, \{X^p_1 \ldots X^p_{n_p}\}) \text{ is a serialization of } dom(O') \text{ against } D'
\end{array}
$$

Figure 6.8: The translation scheme

Thus, we λ-abstract on the corresponding variables, then compute X by applying $\llbracket E \rrbracket .X$ to the parameters $\overline{X_j}$. Since X can depend on itself, this application must be done in a let rec binding over $\overline{X}$. Then, we apply $\llbracket E \rrbracket .Y$ to the parameters that it expects, namely $\overline{Y_i}$, which include $\overline{X}$.

The only operator that remains to be explained is close E. Here, we take advantage of the fact that close removes all input dependencies to generate code that is more efficient than a sequence of freeze operations. We first *serialize* the set of names exported by E against its dependency graph D. That is, we identify strongly connected components of D, then sort them in topological order. The result is an enumeration $(\{X^1_1 \ldots X^1_{n_1}\}, \ldots, \{X^p_1 \ldots X^p_{n_p}\})$ of the exported names where each cluster $\{X^i_1 \ldots X^i_{n_i}\}$ represents mutually recursive definitions, and the clusters are listed in an order such that each cluster depends only on the preceding ones. We then generate a sequence of let rec bindings, one for each cluster, in the order above. In the end, all output components are bound to values with no dependencies, and can be grouped together in a record.

$$\frac{\gamma(x) = 0}{\Gamma \vdash x : \Gamma(x) \ / \ \gamma} \ (\textsc{Var}) \qquad\qquad \Gamma \vdash c : TC(c) \ / \ \gamma \quad (\textsc{Const})$$

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

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

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

$$\frac{\Gamma \vdash M : \tau' \ / \ \gamma' \qquad \Gamma + \{x :_{\_} \tau'\} \vdash N : \tau \ / \ \gamma[x \mapsto d]}{\Gamma \vdash \mathsf{let} \ x = M \ \mathsf{in} \ N : \tau \ / \ \gamma \wedge d \ @ \ \gamma'} \quad (\textsc{Let})$$

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

$$\frac{\forall i : \Gamma \vdash M_i : \tau_i \ / \ \gamma}{\Gamma \vdash \langle \ldots \ X_i = M_i \ \ldots \rangle : \langle \ldots \ X_i :_{\_} \tau_i \ \ldots \rangle \ / \ \gamma} \quad (\textsc{Record})$$

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

Figure 6.9: Typing rules for $\lambda_\circ$

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 $\mathsf{let\ rec}\ x = M\ x\ \mathsf{in}\ \ldots$. In $\lambda_\circ$, 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 $\lambda_\circ$ 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 $\lambda_\circ$ is defined in figure 6.9. Types, written τ, have the following syntax:

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

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

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

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

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

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

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

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

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

6.3.2 Soundness of the target language

To simplify the proofs, we prove the soundness on a subset $\lambda_{\underline{\circ}}$ of $\lambda_\circ$ 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

Degrees

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

Minimum

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

Composition

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

Plus

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

Minus

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

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 $\wedge$, and then $+$ and $-$.

Lemma 18

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

Proof

1. If $d_2 = 0$, we obtain $0 \leq 1$ which is true. If $d_2 = \infty$ we obtain $\infty \leq \infty$. Otherwise, the claim reduces to $d_1 + 1 \leq d_1 + 1$.
2. If $d_3 = 0$, we obtain 0 on both sides of the equality. If $d_3 = \infty$, both sides are equal to ∞. Otherwise we get $d_1 \wedge d_2$ on both sides.
3. If $d_2 = 0$, both sides are equal to 0. If $d_2 = \infty$, then $d_2 \wedge d_3 = d_3$ and $d_1 @ d_2 = \infty$, so both sides are equal to $d_1 @ d_3$. Otherwise, we argue by case on d_3. If $d_3 = 0$, then we obtain 0 on both sides, and if $d_3 = \infty$, we obtain $d_1 @ d_2$ for both sides. Otherwise, $d_2 \wedge d_3 = n \neq 0$, so $d_1 @ (d_2 \wedge d_3) = d_1 = d_1 \wedge d_1 = d_1 @ d_2 \wedge d_1 @ d_3$.
4. If $d_3 = 0$, both sides are equal to 0. If $d_3 = \infty$, we obtain ∞ on both sides. Otherwise, both sides are equal to $d_1 @ d_2$.
5. Both sides reduce to ∞ if $d' = \infty$, to 0 if $d' = 0$, and to $d - 1$ otherwise.
6. By definition of $+$.

7. By definition of $+$ and $-$.
8. By definition of @.
9. By definition of $+$.
10. Since $d+1$ is strictly positive, d' cannot be 0. Thus, $d' = d' - 1 + 1$ by property 7, and the result follows by applying property 9 to $d + 1 \leq d' - 1$.
11. If $d_3 = \infty$ or $d_3 = 0$, both sides reduce to d_3. Otherwise, write $d_3 = n+1$. Then, $d_1 \mathbin{@} d_3 = d_1$ and $d_1 \mathbin{@} d_2 \mathbin{@} d_3 = d_1 \mathbin{@} d_2$, hence it simply remains to prove that $d_1 \leq d_1 \mathbin{@} d_2$. Since $d_2 \geq 1$, we have only two cases: either $d_2 = \infty$, in which case $d_1 \mathbin{@} d_2 = \infty$ which cannot be less than d_1; or $d_2 = m + 1$, in which case $d_1 \mathbin{@} d_2 = d_1$, and the result holds.

□

Lemma 19 *If $\gamma \leq (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$, then there exists γ_1' and γ_2' such that $\gamma = (\gamma_1' - 1) \wedge d \mathbin{@} \gamma_2'$ and $\gamma_1' \leq \gamma_1$ and $\gamma_2' \leq \gamma_2$.*

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

- If $d' = 0$, then we can take $d_1' = d_2' = 0$.
- If $d' = \infty$, then we can take $d_1' = d_1$ and $d_2' = d_2$, because only ∞ is greater than d'.
- If $d' = n + 1$, let $d_1' = n + 2$ and $d_2' = d_2$. By hypothesis we know that $d' \leq d \mathbin{@} d_2$. Since $d_1' - 1 = n + 1 = d'$, we have $(d_1' - 1) \wedge d \mathbin{@} d_2' = d_1' - 1 = d'$. 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.

□

Lemma 20 *If $\gamma \leq (\gamma_1 - 1) \wedge (x \mapsto d)$, then there exists γ_1' such that $\gamma_1' \leq \gamma_1$ and $\gamma = (\gamma_1' - 1) \wedge (x \mapsto d)$.*

Proof We proceed as in the previous proof. Consider a variable y and let $d' = \gamma(y)$ and $d_1 = \gamma_1(y)$. We construct d_1' such that $d_1' \leq d_1$ and $d' = (d_1' - 1) \wedge ((x \mapsto d)(y))$.

- If $d_1 = 0$, then $d_1' = 0$ works.
- Otherwise, we take $d_1' = d' + 1$. This d_1' is suitable because:
  - Since $d' \leq d_1 - 1$, we have $d' + 1 \leq d_1 - 1 + 1$ 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 $d' \leq (d_1 - 1) \wedge (x \mapsto d)(y) \leq (x \mapsto d)(y)$ it follows that $d' \leq (d_1' - 1) \wedge ((x \mapsto d)(y))$.
  - Since $d_1' - 1 = d'$, we have that $(d_1' - 1) \wedge ((x \mapsto d)(y)) \leq d'$.

□

Lemma 21 *Let $n \in \mathbb{N}$. If*

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

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

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

Proof Simply take $\gamma'_0 = \gamma'$ and $\gamma'_i = \gamma_i$ for $i = 1, \ldots, n$. By transitivity we have $\gamma'_0 \leq \gamma_0$ and trivially $\gamma'_i \leq \gamma_i$. It is easy to check that

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

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

$$\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j \geq \gamma' \qquad \text{and} \qquad \bigwedge_{i \in \{1 \ldots n\}} d_i \mathbin{@} \gamma'_i \geq \gamma'$$

hence

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

and the expected equality follows. □

Lemma 22 *If $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$ then there exist γ'_1, γ'_2, d_1, d_2 such that $\gamma_1 = \gamma'_1[x \mapsto d_1]$, $\gamma_2 = \gamma'_2[x \mapsto d_2]$, and $\gamma = (\gamma'_1 - 1) \wedge d_0 \mathbin{@} \gamma'_2$.*

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

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

On $y \neq x$,

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

□

Lemma 23 *If $\gamma[x \mapsto d] = \gamma_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big) \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big)$ then there exist γ'_0 and a γ'_i for each i, such that $\gamma'_0[x \mapsto d_0] = \gamma_0$, $\gamma'_i[x \mapsto d'_i] = \gamma_i$, and $\gamma = \gamma'_0 \wedge \big(\bigwedge_{i,j \in \{1 \ldots n\}} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big) \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big)$, with $d_0 = \gamma_0(x)$ and $d'_i = \gamma_i(x)$ for all i.*

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

Weakening lemmas

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

Lemma 24 (degree restriction) *If $\gamma' \leq \gamma$ and $\Gamma \vdash M : \tau \mathbin{/} \gamma$, then $\Gamma \vdash M : \tau \mathbin{/} \gamma'$.*

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

Rule (app), $M = M_1\ M_2$. By typing hypothesis, we have derivations for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2$, with $\gamma = (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$. By lemma 19, we construct γ_1' and γ_2', such that $\gamma_1' \leq \gamma_1$, $\gamma_2' \leq \gamma_2$ and $\gamma' = (\gamma_1' - 1) \wedge d \mathbin{@} \gamma_2'$. Applying the induction hypothesis twice, we obtain derivations for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1'$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2'$, and we can apply the rule (app) again to obtain the expected result.

Rule (appvar), $M = M_1\ x$. We have a derivation for $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1$ with $\Gamma(x) = \tau'$ and $\gamma = (\gamma_1 - 1) \wedge d$. Hence, $\gamma' \leq (\gamma_1 - 1) \wedge (x \mapsto d)$. Applying lemma 20, we obtain γ_1' such that $\gamma_1' \leq \gamma_1$ and $\gamma' = (\gamma_1' - 1) \wedge (x \mapsto d)$. We can apply rule (appvar) again to derive the expected judgment.

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

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

Using lemma 21, we take $\gamma_N' = \gamma'$ and for all i, $\gamma_i' = \gamma_i$, knowing that $\gamma_N' \leq \gamma_0$ and $\gamma' = \gamma_N' \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i'\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j'\big)$. By induction hypothesis, we know how to derive $\Gamma + \{\dots\ x_j :_{\_} \tau_j\ \dots\} \vdash N : \tau \;/\; \gamma_N'[\dots\ x_j \mapsto d_j\ \dots]$. Hence we can derive $\Gamma \vdash M : \tau \;/\; \gamma'$. □

Lemma 25 (degree weakening) *If* $\Gamma \vdash M : \tau \;/\; \gamma[x \mapsto d]$ *and* $x \notin FV(M)$, *then* $\Gamma \vdash M : \tau \;/\; \gamma$.

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

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

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

Rule (app), $M = M_1\ M_2$. We have $\Gamma \vdash M_1 : \tau' \xrightarrow{d_0} \tau \;/\; \gamma_1$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge d_0 \mathbin{@} \gamma_2$. Applying lemma 22, we obtain d_1, d_2, γ_1' and γ_2' such that $\gamma = (\gamma_1' - 1) \wedge d_0 \mathbin{@} \gamma_2'$, $\gamma_1'[x \mapsto d_1] = \gamma_1$ and $\gamma_2'[x \mapsto d_2] = \gamma_2$. By induction hypothesis we can derive $\Gamma \vdash M_1 : \tau' \xrightarrow{d_0} \tau \;/\; \gamma_1'$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2'$. The expected result follows by rule (app).

Rule (appvar), $M = M_1 y$, with $y \neq x$ by hypothesis $x \notin FV(M)$. We have a derivation of $\Gamma \vdash M_1 : \tau_1 \xrightarrow{d_0} \tau_2 \;/\; \gamma_1$ with $\gamma[x \mapsto d] = (\gamma_1 - 1) \wedge (y \mapsto d_0)$. Take $\gamma_1' = \gamma_1[x \mapsto \gamma(x) + 1]$.

We have $\gamma_1'[x \mapsto \gamma_1(x)] = \gamma_1$ and $\gamma = (\gamma_1' - 1) \wedge (y \mapsto d_0)$. The first equality is straightforward, and the second equality follows from the facts that $\gamma(x) = \gamma(x) + 1 - 1$, and for any $z \neq x$, $((\gamma_1 - 1) \wedge (y \mapsto d_0))(z) = ((\gamma_1' - 1) \wedge (y \mapsto d_0))(z)$. We then conclude by induction hypothesis as above.

Rule (rec), $M = \mathsf{let\ rec}\ \ldots\ x_i = M_i\ \ldots\ \mathsf{in}\ N$. 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, $d_{ik} \leq d_{ij} @ d_{jk}$ and for all i, j, $d_{ij} \geq 1$ and $\gamma[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i @ \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i @ d_{ij} @ \gamma_j\big)$. Lemma 23 shows the existence of γ_N' and γ_i' for all i such that $\gamma_N'[x \mapsto d_N] = \gamma_N$, and for all i $\gamma_i'[x \mapsto d_i'] = \gamma_i$, and $\gamma = \gamma_N' \wedge \big(\bigwedge_i d_i @ \gamma_i'\big) \wedge \big(\bigwedge_{i,j} d_i @ d_{ij} @ \gamma_j'\big)$, with $d_N = \gamma_N(x)$ and for all i, $d_i' = \gamma_i'(x)$. Applying the induction hypothesis, we derive

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

and for all i

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

The result follows by rule (rec). □

Lemma 26 (type weakening) *If $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau\ /\ \gamma$ and $x \notin FV(M)$, then $\Gamma \vdash M : \tau\ /\ \gamma$.*

Proof Straightforward by induction on the typing derivation. □

Substitution lemmas

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

Lemma 29 (substitution) *If $\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau\ /\ \gamma_1[x \mapsto d]$, and $\Gamma \vdash M_2 : \tau'\ /\ \gamma_2$, with $x \notin FV(M_2) \cup dom(\gamma_2)$, then $\Gamma \vdash M_1\{x\}M_2 : \tau\ /\ \gamma_1 \wedge d @ \gamma_2$.*

Proof We proceed by induction on the typing derivation of M_1 and case analysis on the last typing rule used. We write $M = M_1\{x\}M_2$, $\Gamma' = \Gamma + \{x \mapsto \tau'\}$, and $\gamma_0 = \gamma_1 \wedge d @ \gamma_2$.

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 $\gamma_1 \wedge 0 @ \gamma_2 \leq \gamma_2$, which is true by lemma 18.

If $y \neq x$, then $x \notin FV(M)$ and $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau\ /\ \gamma_1[x \mapsto d]$, so by lemmas 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

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

with $\tau = \tau_1 \xrightarrow{d_0} \tau_2$ and $\gamma_3[y\mapsto d_0] = (\gamma_1[x\mapsto d]-1)[y\mapsto d_0] = (\gamma_1 - 1)[x\mapsto(d-1); y\mapsto d_0]$. Take $M_3' = M_3\{x\}M_2$. By induction hypothesis, we have $\Gamma + \{y \mapsto \tau_1\} \vdash M_3' : \tau_2 \ / \ (\gamma_1 - 1)[y\mapsto d_0] \wedge (d-1) @ \gamma_2$. Since y is fresh, it does not occur in γ_2, hence

$$\begin{array}{l}
(\gamma_1 - 1)[y \mapsto d_0] \wedge (d-1) @ \gamma_2 \\
= ((\gamma_1 - 1) \wedge (d-1) @ \gamma_2)[y \mapsto d_0] \\
= ((\gamma_1 - 1) \wedge (d @ \gamma_2 - 1))[y \mapsto d_0] \text{ by lemma 18} \\
= ((\gamma_1 \wedge d @ \gamma_2) - 1)[y \mapsto d_0] = (\gamma_0 - 1)[y \mapsto d_0]
\end{array}$$

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

Rule (app), $M_1 = M_3 \ M_4$. We have $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3$ and $\Gamma' \vdash M_4 : \tau'' \ / \ \gamma_4$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge d_0 @ \gamma_4$. By lemma 22, if $d_3 = \gamma_3(x)$ and $d_4 = \gamma_4(x)$, there exists γ_3' and γ_4' such that $\gamma_3'[x \mapsto d_3] = \gamma_3$, $\gamma_4'[x \mapsto d_4] = \gamma_4$, and $\gamma_1 = (\gamma_3' - 1) \wedge d_0 @ \gamma_4'$. By induction hypothesis, if $M_3' = M_3\{x\}M_2$ and $M_4' = M_4\{x\}M_2$, then $\Gamma \vdash M_3' : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3' \wedge d_3 @ \gamma_2$ 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

$$\begin{array}{rl}
\multicolumn{2}{l}{(\gamma_3' - 1) \wedge (d_3 @ \gamma_2 - 1) \wedge (d_0 @ \gamma_4') \wedge (d_0 @ d_4 @ \gamma_2)} \\
= & \gamma_1 \wedge (d_3 @ \gamma_2 - 1) \wedge (d_0 @ d_4 @ \gamma_2) \\
= & \gamma_1 \wedge ((d_3 - 1_{\wedge} d_0 @ d_4) @ \gamma_2 \\
= & \gamma_1 \wedge d @ \gamma_2 \\
= & \gamma_0
\end{array}$$

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 $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3$ (*) and $\Gamma'(y) = \Gamma'(x) = \tau' = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Take $\gamma_3' = \gamma_3[x \mapsto \gamma_1(x) + 1]$. We have $\gamma_1 = (\gamma_3' - 1)$ and $\gamma_3'[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we can write the premise (*) as follows

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

n Hence, by induction hypothesis we have

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

with $d_3 = \gamma_3(x)$. Then by rule (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 $\Gamma' \vdash M_3 : \tau'' \xrightarrow{d_0} \tau \ / \ \gamma_3$ (*) and $\Gamma'(y) = \Gamma(y) = \tau''$ and $\gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0)$. Take

$\gamma_3' = \gamma_3[x \mapsto \gamma_1(x)+1]$. We have $\gamma_1 = (\gamma_3' - 1) \wedge (y \mapsto d_0)$, and $\gamma_3'[x \mapsto \gamma_3(x)] = \gamma_3$. Thus we rewrite the premise (*) as follows:

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

By induction hypothesis, it follows that

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

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

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

which yields by lemma 18

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

Moreover,

$$\begin{array}{rcl}
\gamma_0 & = & \gamma_1 \wedge d \mathbin{@} \gamma_2 \\
& = & (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge d \mathbin{@} \gamma_2 \\
& = & (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge (d_3 - 1) \mathbin{@} \gamma_2 \\
\multicolumn{3}{l}{\text{(because } \gamma_1[x \mapsto d] = (\gamma_3 - 1) \wedge (y \mapsto d_0))} \\
& = & (\gamma_3' - 1) \wedge (y \mapsto d_0) \wedge (d_3 \mathbin{@} \gamma_2 - 1) \qquad \text{(by lemma 18)}
\end{array}$$

Thus, the expected result holds.

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

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

We write $N' = N\{x\}M_2$ and for all i, $N_i' = N_i\{x\}M_2$. We have $\gamma_1[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \delta_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j\big)$. Lemma 23 shows that we can construct γ_N' and a δ_i' for all i such that $\gamma_N'[x \mapsto d_N] = \gamma_N$, and $\delta_i'[x \mapsto d_i^0] = \delta_i$ for all i and $\gamma_1 = \gamma_N' \wedge \big(\bigwedge_i d_i \mathbin{@} \delta_i'\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \delta_j'\big)$, with $d_N = \gamma_N(x)$ and $d_i^0 = \delta_i(x)$ for each i. Thus, the two premises can be rewritten as follows:

$$\begin{array}{c}
\Gamma' + \{\ldots\ x_j :_{\_} \tau_j\ \ldots\} \vdash N : \tau \; / \; \gamma_N'[\ldots\ x_j \mapsto d_j\ \ldots][x \mapsto d_N] \\
\text{for all } i,\ \Gamma' + \{\ldots\ x_j :_{\_} \tau_j\ \ldots\} \vdash N_i : \tau_i \; / \; \delta_i'[\ldots\ x_j \mapsto d_{ij}\ \ldots][x \mapsto d_i^0]
\end{array}$$

By induction hypothesis, it follows that

$$\begin{array}{c}
\Gamma + \{\ldots\ x_j :_{\_} \tau_j\ \ldots\} \vdash N' : \tau \; / \; \gamma_N'[\ldots\ x_j \mapsto d_j\ \ldots] \wedge d_N \mathbin{@} \gamma_2 \\
\text{for all } i,\ \Gamma + \{\ldots\ x_j :_{\_} \tau_j\ \ldots\} \vdash N_i' : \tau_i \; / \; \delta_i'[\ldots\ x_j \mapsto d_{ij}\ \ldots] \wedge d_i^0 \mathbin{@} \gamma_2
\end{array}$$

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

$$\Gamma \vdash M : \tau \; / \; \gamma_N' \wedge d_N \mathbin{@} \gamma_2 \wedge \bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} (\delta_j' \wedge d_j^0 \mathbin{@} \gamma_2) \wedge \bigwedge_i d_i \mathbin{@} (\delta_i' \wedge d_i^0 \mathbin{@} \gamma_2)$$

According to lemma 18, the degree environment above is equal to

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

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

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

we know that

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

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

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

This completes the proof. □

We now extend the previous lemma to the case of parallel substitution, exploiting the fact that $M\{\ldots\ x_i \mapsto M_i\ \ldots\}$ is equal to $M\{x_1\}y_1\ \ldots\ \{x_n\}y_n\{y_1\}M_1\ \ldots\ \{y_n\}M_n$, 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* $y \notin FV(M)$, *then* $\Gamma + \{y :_{\_} \tau\} \vdash M\{x\}y : \tau \;/\; \gamma[y \mapsto d]$.

Proof Easy induction on the typing derivation of M. □

Lemma 31 (parallel substitution) *Assume* $\Gamma + \{\ldots\ x_i :_{\_} \tau_i\ \ldots\} \vdash M : \tau \;/\; \gamma_M[\ldots\ x_i \mapsto d_i\ \ldots]$, *and for all* $j \in \{1 \ldots n\}$, $\Gamma \vdash M_j : \tau_j \;/\; \gamma_j$ *with for all* i, j, $x_i \notin FV(M_j) \cup dom(\gamma_j)$. *Then,* $\Gamma \vdash M\{\ldots\ x_i \mapsto M_i\ \ldots\} : \tau \;/\; \gamma_M \wedge \bigwedge_i d_i \mathbin{@} \gamma_i$.

Proof Write $M\{\ldots\ x_i \mapsto M_i\ \ldots\}$ as $M\{x_1\}y_1\ \ldots\ \{x_n\}y_n\{y_1\}M_1\ \ldots\ \{y_n\}M_n$ where the y_i are fresh. We first apply lemma 27 n times to obtain $\Gamma + \{\ldots\ y_i :_{\_} \tau_i\ \ldots\} \vdash M\{x_1\}y_1\ \ldots\ \{x_n\}y_n : \tau \;/\; \gamma_M[\ldots\ y_i \mapsto d_i\ \ldots]$. We then apply lemma 29 n times again, successively using the n typing hypotheses for the M_i. This leads to the desired judgment. □

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 occurences of y must be taken into account, which is done through the $\wedge$ operation.

Lemma 30 (substitution by a variable) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau \;/\; \gamma[x \mapsto d]$ *and* $\Gamma(y) = \tau'$, *then* $\Gamma \vdash M\{x\}y : \tau \;/\; \gamma \wedge (y \mapsto d)$.

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

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

which yields

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

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

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

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

By induction hypothesis, we can derive $\dfrac{\Gamma \vdash M_1\{x\}y : \tau'' \xrightarrow{d_0} \tau \;/\; \gamma_1' \wedge (y \mapsto \gamma_1(x)) \qquad \Gamma \vdash M_2\{x\}y : \tau'' \;/\; \gamma_2' \wedge (y \mapsto \gamma_2(x}{\Gamma \vdash M' : \tau \;/\; (\gamma_1' - 1) \wedge (y \mapsto (\gamma_1(x) - 1)) \wedge d_0 \;@\; (\gamma_2' \wedge (y \mapsto \gamma_2(x)))}$
The degree environment in the conclusion is equal to

$$(\gamma_1' - 1) \wedge d_0 \;@\; \gamma_2' \wedge (y \mapsto ((\gamma_1(x) - 1) \wedge d_0 \;@\; \gamma_2(x))) = \gamma \wedge (y \mapsto d)$$

The desired result follows.

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

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

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

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

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

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

Case $z \neq x$. Define $\gamma'_1 = \gamma_1[x \mapsto \gamma(x) + 1]$. We have $\gamma = (\gamma'_1 - 1) \wedge (z \mapsto d_0)$ and $\gamma'_1[x \mapsto \gamma_1(x)] = \gamma_1$. By induction hypothesis, we obtain

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

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

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

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

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

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

Moreover, $\gamma[x \mapsto d] = \gamma_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma_j\big)$. By lemma 23, we can construct γ'_N and γ'_i for each i satisfying the following conditions: $\gamma = \gamma'_N \wedge \big(\bigwedge_i d_i \mathbin{@} \gamma'_i\big) \wedge \big(\bigwedge_{i,j} d_i \mathbin{@} d_{ij} \mathbin{@} \gamma'_j\big)$, $\gamma_N = \gamma'_N[x \mapsto d_N]$, and for all i, $\gamma_i = \gamma'_i[x \mapsto d'_i]$, with $d_N = \gamma_N(x)$ and for all i, $d'_i = \gamma_i(x)$. Applying the induction hypothesis, we obtain derivations for the following judgments:

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

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

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

This concludes the proof. □

Soundness

The soundness of $\lambda_\circ$'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 $\mathsf{let\ rec}$ term.

Lemma 28 *Assume* $\Gamma + \{\ldots\ x_i :_{\_} \tau_i\ \ldots\} \vdash M_j : \tau_j \ /\ \gamma_j[\ldots\ x_i \mapsto d_{ji}\ \ldots]$ *for all* $j \in \{1 \ldots n\}$. *Further assume that for all* i, j, $d_{ij} \geq 1$ *and for all* i, j, k, $d_{ik} \leq d_{ij} \mathbin{@} d_{jk}$. *Then, for any* $i_0 \in \{1 \ldots n\}$,

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

Proof By application of rule (rec), we obtain

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

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

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

and the expected result follows. □

Lemma 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

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

Moreover, $\gamma = (\gamma_0 - 1) \wedge (y \mapsto d)$ and $M' = M_1\{x\}y$. By lemma 30, we have

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

which is the expected result.

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

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

Moreover, $M' = M_1\{x\}v$ and $\gamma = (\gamma_1 - 1) \wedge d \mathbin{@} \gamma_2$. By lemma 29, it follows that $\Gamma \vdash M' : \tau \ /\ \gamma$, as expected.

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

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

By lemma 28, it follows that

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

By lemma 31, we obtain

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

which is identical to the expected result

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

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

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 $M \longrightarrow M'$.*

Proof The proof is a standard inductive argument on the typing derivation of M, and case analysis on the last typing rule used.

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

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

Rule (app), $M = M_1\ M_2$. We have $\Gamma \vdash M_1 : \tau' \xrightarrow{d} \tau \;/\; \gamma_1$ and $\Gamma \vdash M_2 : \tau' \;/\; \gamma_2$. Moreover, $\gamma = (\gamma_1 - 1) \wedge d \,@\, \gamma_2$.

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 $\square\ M_2$. If M_1 is a value and M_2 reduces, M also reduces via the context $M_1\ \square$. If both M_1 and M_2 are values, the type $\tau' \xrightarrow{d} \tau$ of M_1 guarantees that M_1 is either a variable or an abstraction. But M_1 cannot be a variable, because $\gamma \geq 1$ implies $\gamma_1 \geq 2$. Hence, M_1 is an abstraction and we can apply the (beta) rule to reduce M.

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

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

Theorem 3 (soundness of $\lambda_\circ$) *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)
- $\mathit{FCT}_D(X,I) = (M_1^{v_1}, \ldots, M_n^{v_n})$, for $\mathit{Pred}(D) \subset \mathit{dom}(I)$, where
  - $D^{-1}(X) = (X_1, \ldots, X_n)$
  - for all $i \in \{1 \ldots n\}$, $I(X_i) = M_i$ and $D(X_i, X) = v_i$.
- $\mathit{Pred}(D) = \{X \mid X \xrightarrow{v} Y \in D, X, Y \in \mathit{Names}, v \in \mathit{Vals}\}$
- $\mathit{Succ}(D) = \{Y \mid X \xrightarrow{v} Y \in D, X, Y \in \mathit{Names}, v \in \mathit{Vals}\}$

Figure 6.11: Operations on graphs

$$\begin{array}{rcl}
[\![\tau_1 \to \tau_2]\!] & = & \tau_1 \xrightarrow{0} \tau_2 \\
[\![\mathtt{int}]\!] & = & \mathtt{int} \\
[\![\mathtt{bool}]\!] & = & \mathtt{bool} \\
[\![\{I;O;D\}]\!] & = & \langle X :_{\_} [\![O(X)]\!]_{X,D,I} \mid X \in \mathit{dom}(O)\rangle \text{ if } \vdash \{I;O;D\} \\
[\![M]\!]_{X,D,I} & = & [\![M_1]\!] \xrightarrow{v_1+(n-1)} [\![M_2]\!] \xrightarrow{v_2+(n-2)} \ldots\ [\![M_n]\!] \xrightarrow{v_n} [\![M]\!] \\
 & & \text{where } (M_1^{v_1}, \ldots, M_n^{v_n}) = \mathit{FCT}_D(X,I)
\end{array}$$

Figure 6.12: Translation of types

Lemma 29 (substitution) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M_1 : \tau\ /\ \gamma_1[x \mapsto d]$, *and* $\Gamma \vdash M_2 : \tau'\ /\ \gamma_2$, *with* $x \notin \mathit{FV}(M_2) \cup \mathit{dom}(\gamma_2)$, *then* $\Gamma \vdash M_1\{x\}M_2 : \tau\ /\ \gamma_1 \wedge d\ @\ \gamma_2$.

Lemma 30 (substitution by a variable) *If* $\Gamma + \{x \mapsto \tau'\} \vdash M : \tau\ /\ \gamma[x \mapsto d]$ *and* $\Gamma(y) = \tau'$, *then* $\Gamma \vdash M\{x\}y : \tau\ /\ \gamma \wedge (y \mapsto d)$.

Lemma 31 (parallel substitution) *If* $\Gamma + \{\ldots\ x_i :_{\_} \tau_i\ \ldots\} \vdash M : \tau\ /\ \gamma_M[\ldots\ x_i \mapsto d_i\ \ldots]$, *and for all* $j \in \{1 \ldots n\}$, $\Gamma \vdash M_j : \tau_j\ /\ \gamma_j$ *with for all* i, j, $x_i \notin \mathit{FV}(M_j) \cup \mathit{dom}(\gamma_j)$, *then* $\Gamma \vdash M\{\ldots\ x_i \mapsto M_i\ \ldots\} : \tau\ /\ \gamma_M \wedge \bigwedge_i d_i\ @\ \gamma_i$.

The soundness of $\lambda_\circ$ 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* $\Gamma \vdash M : \tau\ /\ \gamma$ *and* $M \longrightarrow M'$, *then* $\Gamma \vdash M : \tau\ /\ \gamma$.

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* $M \longrightarrow M'$.

□

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 MM_e expression translates to a well-typed $\lambda_\circ$ expression. The soundness of $\lambda_\circ$ then ensures that the translation evaluates correctly.

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

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 $\lambda_\circ$ types. We start by defining useful operations on graphs and signatures in figure 6.11. We define $FCT_D(X, I)$ as the list of the types and valuations of the predecessors of X in D according to I, ordered lexicographically. Then, $Pred(D)$ and $Succ(D)$ 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 $[\![\Gamma]\!] = [\![\cdot]\!] \circ \Gamma$. Moreover, we define the initial degree environment corresponding to a type environment as $d^o(\Gamma) = \underline{0} \circ \Gamma$, that is to say the function equal to 0 on $dom(\Gamma)$ and ∞ elsewhere. In the sequel, we will often use valuations as degrees. It is worth noticing that for all valuations v_1, and v_2, $\min(v_1, v_2) = v_1 \wedge v_2 = v_1 \mathbin{@} v_2$.

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 $\lambda_\circ$-term.

We start by stating three typing rules that are admissible in $\lambda_\circ$, 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 $\lambda_\circ$.*

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

Lemma 35 (n abstractions) *The following typing rule is admissible for the type system of $\lambda_\circ$.*

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

Lemma 36 (n applications) *The following typing rule is admissible for the type system of $\lambda_\circ$.*

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

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

Lemma 37 (translation of close) *Assume $\Gamma|- e : [\![\{I;O;D\}]\!] \;/\; d^o(\Gamma)$. Let $X_1, \ldots, X_n$ be names such that $\overline{X_i} \notin dom(\Gamma)$ and $O(X_i) = I(X_i)$ and $D(X_i, X_j) \neq 0$ for $i, j \in \{1, \ldots, n\}$. Further assume that for all immediate predecessors X of one of the X_i in D, either X is one of the X_i, or $\Gamma(\overline{X}) = I(X)$. Let M be an expression and τ be a type such that $\Gamma'|- M : \tau \;/\; d^o(\Gamma')$, where $\Gamma' = \Gamma + \{\overline{X_1} : O(X_1), \ldots, \overline{X_n} : O(X_n)\}$. Then,*

$$\Gamma|- \mathsf{let\ rec}\ \overline{X_1} = e.X_1\ \overline{D^{-1}(X_1)}\ \mathsf{and}\ \ldots\ \mathsf{and}\ \overline{X_n} = e.X_n\ \overline{D^{-1}(X_n)}\ \mathsf{in}\ M : \tau \;/\; d^o(\Gamma)$$

Proof By definition of the translation of a mixin signature, and the hypotheses on Γ, the conditions of lemma 36 are met, and we obtain

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

Since $\overline{X_j} \notin dom(\Gamma)$ for all j, the degree environment above is pointwise greater or equal to $d^o(\Gamma)[\overline{X_j} \mapsto D(X_j, X_i) \mid j \in \{1, \ldots, n\}]$. Thus, by lemma 24, it follows that

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

Moreover, $D(X_j, X_i) \in \{1, \infty\}$ for all i and j. Hence, the premises of the (rec) typing rule are met. Applying the weakening lemma 24 to its conclusion, we obtain the desired result. □

Lemma 38 (translation of freeze) *Assume $\Gamma|- e : [\![\{I;O;D\}]\!]/d^o(\Gamma)$, 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 $I' = I_{\setminus X}$. Then, for all names $Y \in dom(O)$, if $X \notin D^{-1}(Y)$ we have*

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

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

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

Proof Recall the definition of D':

$$D' = D!X = (D \cup D_{around}) \setminus D_{remove}$$

where $D_{around} = \{Z \xrightarrow{v_1' \wedge v_2'} Y \mid (Z \xrightarrow{v_1} X) \in D,\ (X \xrightarrow{v_2} Y) \in D\}$ and $D_{remove} = \{X \xrightarrow{v} Y \mid Y \in Names,\ v \in \{0, 1\}\}$.

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 $[\![O(X)]\!]_{X,D!X,I_{\setminus X}} = [\![O(X)]\!]_{X,D,I}$ 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 $\Gamma' = \Gamma + \{\overline{Z} : [\![I(Z)]\!] \mid Z \in D^{-1}(Y)\}$. By lemma 36, and using the fact that e is not one of the $\overline{Z}$, it follows that

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

and

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

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 $\Gamma'|- \mathsf{let\ rec}\ \overline{X} = e.X\ \overline{D^{-1}(X)}\ \mathsf{in}\ e.Y\ \overline{D^{-1}(Y)} : [\![O(Y)]\!] \;/\; \gamma$ where

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

By definition of $D' = D!X$, γ is equal to $\{e\mapsto 0; \overline{Z}\mapsto D'(Z,Y) \mid Z \in D'^{-1}(Y)\}$. Applying lemma 35, we obtain

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

which implies the desired result by weakening. □

Theorem 4 (soundness of the translation) *If $\Gamma \vdash E : M$, then $[\![\Gamma]\!] \vdash [\![\overline{E}]\!] : [\![M]\!]\ /\ d^o(\Gamma) + IsRec(E)$.*

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

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

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

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

Let $o = X_i \overset{i\in I}{\mapsto} E_i$, $O = X_i \overset{i\in I}{\mapsto} M_i$, $v_i = IsRec(E_i)$ and $\iota = y_j \overset{j\in J}{\mapsto} Y_j$, with $I(Y_j) = M'_j$ for all j, with the X_is and Y_js ordered lexicographically, that is, if $i_1 < i_2$, then $X_{i_1} <_{lex} X_{i_2}$, and similarly for the Y_js.

By induction hypothesis, for all i, we have $[\![\Gamma]\!] + [\![I \circ \iota]\!] \vdash [\![\overline{E_i}]\!] : [\![M_i]\!]\ /\ d^o(\Gamma + I \circ \iota) + v_i$.

But $FV([\![\overline{E_i}]\!]) = FV(E_i)$ and $FV(E_i) \cap dom(\iota) = \iota^{-1}(D^{-1}(X_i))$, so we can apply lemma 35, and weakening lemmas 25 and 26 to eliminate variables of $dom(\iota)$ 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

$$\Gamma \vdash \vec{\lambda}\iota^{-1}(D^{-1}(X_i)).[\![\overline{E_i}]\!] : [\![M''_1]\!] \xrightarrow{v_i+(n-1)} \ldots [\![M''_n]\!] \xrightarrow{v_i} [\![M_i]\!]\ /\ d^o(\Gamma)$$

But $[\![M_i]\!]_{X_i,D,I} = [\![M''_1]\!] \xrightarrow{v_i+(n-1)} \ldots [\![M''_n]\!] \xrightarrow{v_i} [\![M_i]\!]$, because $D(Z_k, X_i) = \nu(\iota^{-1}(Z_k),\ E_i) = IsRec(E_i) = v_i$. The desired result follows.

Closing: $E = \mathsf{close}\,E'$ 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: $E = E_1\ !\ X$. 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. □

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

Theorem 4 (soundness of the translation) *If* $\Gamma \vdash E : M$*, then* $[\![\Gamma]\!] \vdash [\![\overline{E}]\!] : [\![M]\!] \ / \ d^o(\Gamma) + \mathit{IsRec}(E)$.

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

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 $x_1 = e_1 \dots x_n = e_n$. 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 = λ x.(... f ...)` is accepted, since no computation will try to use the value of f. Contrarily, a recursive definition like `f = (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 adresses 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 v_1 computed for e_1, for example if $e_2 = (x_1\, 1)$, 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,

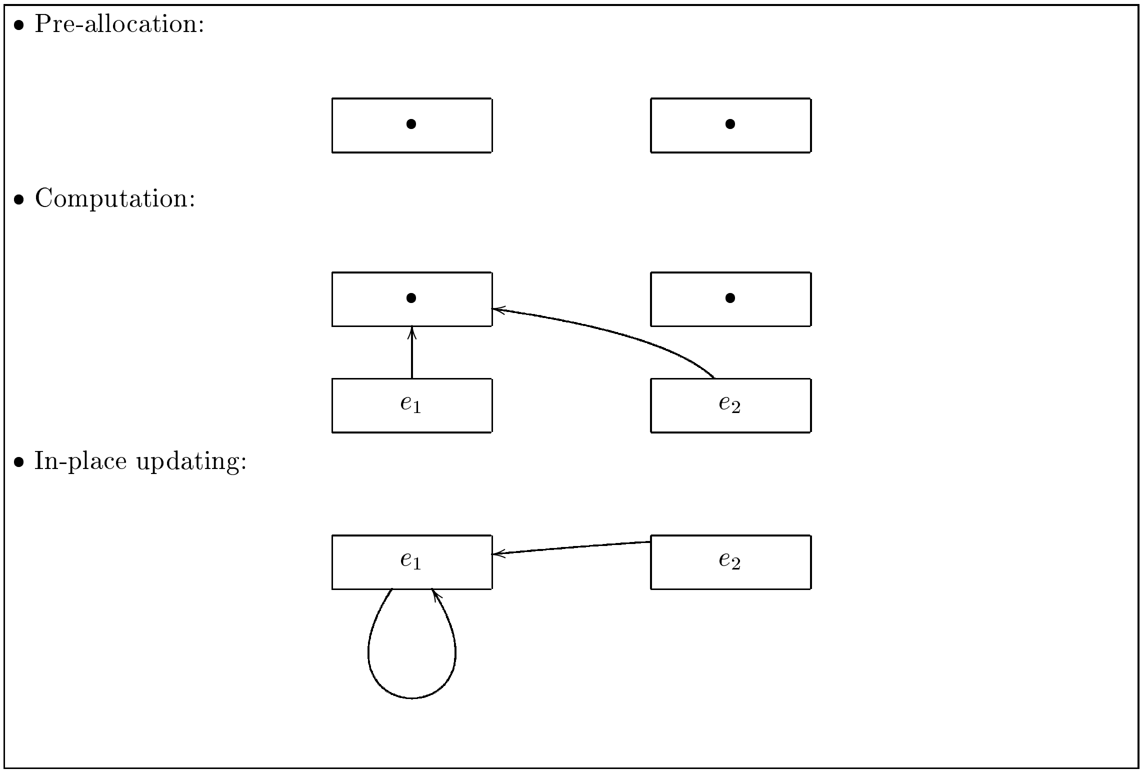


Figure 7.1: The “in-place updating trick”

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 $\lambda_\circ$

7.2.1 Syntax

The syntax of $\lambda_\circ$ 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 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

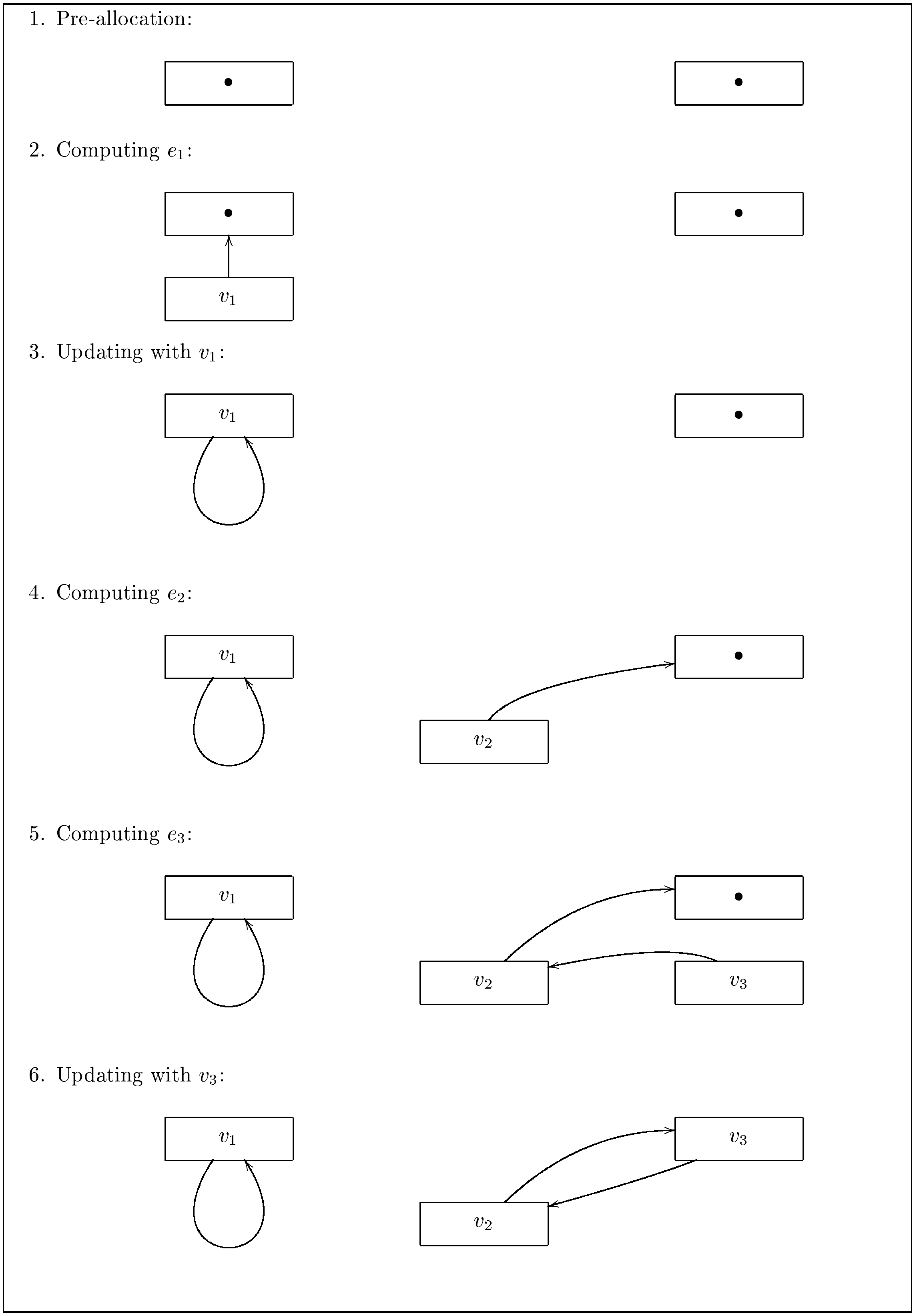


Figure 7.2: The refined “in-place updating trick”

$$\begin{array}{rrll} & x & \in \mathit{Vars} & \text{Variable} \\ & X & \in \mathit{Names} & \text{Name} \\ \text{Expression:} & e \in \mathit{expr} & ::= x \mid \lambda x.e \mid e_1 e_2 & \\ & & \mid \{X_1 = e_1 \dots X_n = e_n\} \mid e.X & \\ & & \mid \mathsf{let\ rec}\ \ x_1 = e_1 \dots x_n = e_n\ \ \mathsf{in}\ \ e & \end{array}$$

Figure 7.3: Syntax of $\lambda_\circ$

- More meta-variables:

$$\begin{array}{ll} s ::= X_1 = e_1 \dots X_n = e_n & \text{Record} \\ b ::= x_1 = e_1 \dots x_n = e_n & \text{Binding} \end{array}$$

- Notations:

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

- Expressions of predictable shape:

$e_\downarrow \in \mathit{Predictable} ::= \{o\} \mid \langle \iota; o \rangle \mid \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e_\downarrow$

Figure 7.4: Meta-variables and notations

recursive definition has the shape $\mathsf{let\ rec}\ \ x_1 = e_1 \dots x_n = e_n\ \ \mathsf{in}\ \ e$, where arbitrary expressions are syntactically allowed as the right-hand side of a definition.

Syntactic correctness Records $s = (X_1 = e_1 \dots X_n = e_n)$ and bindings $b = (x_1 = e_1 \dots x_n = e_n)$ 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 $b = (x_1 = e_1 \dots x_n = e_n)$. 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 $e_\downarrow \in \mathit{Predictable} ::= \{o\} \mid \langle \iota; o \rangle \mid \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e_\downarrow$.

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 *dom*, the codomain *cod*, the restriction $\cdot_{|P}$ to a set P, or the co-restriction $\cdot_{\backslash P}$ 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

$$\begin{array}{rcl}
\mathit{UnsafeNewNames}(x, \lambda x.e) & = & \mathit{Capt}_x(e) \cup \mathit{FV}(e) \\
\mathit{UnsafeNewNames}(x, \textsf{let rec } b \textsf{ in } e) & = & \big(\ \ \mathit{FV}(\textsf{let rec } b \textsf{ in } e) \\
 & & \cup\ \ \mathit{Capt}_x(e) \\
 & & \cup\ \ \displaystyle\bigcup_{(y \diamond f) \in b} (\{y\} \cup \mathit{Capt}_x(f))\big) \\
 & & \setminus\ \ \{x\}
\end{array}$$

Figure 7.5: Unsafe new names in $\lambda_\circ$

$$\begin{array}{rcl}
\mathit{Capt}_x(\textsf{let rec } b \textsf{ in } e) & = & \begin{cases} \displaystyle\bigcup_{y \in dom(b)} (\{y\} \cup \mathit{Capt}_x(b(y))) \cup \mathit{Capt}_x(e) & \\ \qquad\qquad \text{if } x \in \mathit{FV}(\textsf{let rec } b \textsf{ in } e) & \\ \emptyset \qquad\qquad\qquad \text{otherwise} & \end{cases} \\
\mathit{Capt}_x(x) = \mathit{Capt}_x(\mathsf{c}) & = & \emptyset \\
\mathit{Capt}_x(\lambda y.e) & = & \begin{cases} \{y\} \cup \mathit{Capt}_x(e) & \\ \quad \text{if } x \in \mathit{FV}(\lambda y.e) & \\ \emptyset \qquad \text{otherwise} & \end{cases}
\end{array}$$

Other cases easy.

Figure 7.6: Capture in $\lambda_\circ$

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 *Capt* function in figure 7.6. Basically, $\mathit{Capt}_x(e)$ denotes the set of binding variables located above occurrences of x in e. For instance $\mathit{Capt}_x(\lambda y.x)$ is the set $\{y\}$. A new name can also be unsafe for a binder when it is free in the considered sub-expression. For example, renaming x to y in $\lambda x.(xy)$ does not preserve meaning. The structural contraction relation, $\leadsto_s$, 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 $\dashrightarrow_s$ is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure ${\dashrightarrow_s}^*$ of $\dashrightarrow_s$ 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 $\sigma = \{x \mapsto y\}$.

$$\begin{array}{rcl}
x\{\sigma\} & = & y \\
z\{\sigma\} & = & z \text{ if } z \neq x \\
\{X_1 = e_1 \ldots X_n = e_n\}\{\sigma\} & = & \{X_1 = e_1\{\sigma\} \ldots X_n = e_n\{\sigma\}\} \\
(\lambda z.e)\{\sigma\} & = & \begin{cases} \lambda z.(e\{\sigma\}) & \text{if } z \notin \{x, y\} \\ \lambda z.e & \text{otherwise} \end{cases} \\
(\textsf{let rec } b \textsf{ in } e)\{\sigma\} & = & \begin{cases} \textsf{let rec } b\{\sigma\} \textsf{ in } e\{\sigma\} & \text{if } \{x, y\} \cap dom(b) = \emptyset \\ \textsf{let rec } b \textsf{ in } e & \text{otherwise} \end{cases} \\
(x_1 = e_1 \ldots x_n = e_n)\{\sigma\} & = & (x_1 = e_1\{\sigma\} \ldots x_n = e_n\{\sigma\})
\end{array}$$

Other cases easy.

Figure 7.7: Variable renaming in $\lambda_\circ$

$$\frac{y \notin \mathit{UnsafeNewNames}(x, \mathsf{let\ rec}\ \ b_1, x = e, b_2\ \ \mathsf{in}\ \ f) \qquad \sigma = x \mapsto y}{\mathsf{let\ rec}\ \ b_1, x = e, b_2\ \ \mathsf{in}\ \ f \rightsquigarrow_s \mathsf{let\ rec}\ \ b_1\{\sigma\}, y = e\{\sigma\}, b_2\{\sigma\}\ \ \mathsf{in}\ \ f\{\sigma\}}$$

$$\frac{y \notin \mathit{UnsafeNewNames}(x, \lambda x.e)}{\lambda x.e \rightsquigarrow_s \lambda y.(e\{x \mapsto y\})}$$

Figure 7.8: Structural contraction relation of $\lambda_\circ$

Configuration: $c ::= b \vdash e$

Value: $v \in \mathit{values} ::= x \mid \lambda x.e \mid \{s_v\}$

Answer: $a \in \mathit{answers} ::= b_v \vdash v$

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 $\lambda_\circ$

7.2.3 Semantics

The semantics of $\lambda_\circ$ 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 $\{s_v\}$, 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 7.10, 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.

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 $e_1 + (\mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e_2)$ becomes $\mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ e_1 + e_2$.

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 $\mathsf{let\ rec}\ \ b_1, x = (\mathsf{let\ rec}\ \ b_2\ \ \mathsf{in}\ \ e), b_3\ \ \mathsf{in}\ \ f$ becomes $\mathsf{let\ rec}\ \ b_1, b_2, x = e, b_3\ \ \mathsf{in}\ \ f$, provided no variable capture occurs.

3. A third operation is external merging. The shape of results in $\lambda_\circ$ 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 $\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ \mathsf{let\ rec}\ \ b_2\ \ \mathsf{in}\ e$ becomes $\mathsf{let\ rec}\ \ b_1, b_2\ \ \mathsf{in}\ \ e$, 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 $\mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ \mathbb{C}\,[x]$ becomes $\mathsf{let\ rec}\, b\, \mathsf{in}\, \mathbb{C}\,[e]$, if $x = e$ appears in b and x is not captured by $\mathbb{C}$, 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 $\mathsf{let\ rec}\ \ b_1, y = \mathbb{C}\,[x], b_2\ \ \mathsf{in}\ \ e$ becomes $\mathsf{let\ rec}\, b_1, y = \mathbb{C}\,[f], b_2\, \mathsf{in}\, e$ if x is defined as f in b_1, and not captured by $\mathbb{C}$, and no variable capture occurs.

- **Contraction rules**

$$\{X_1 = v_1 \ldots X_n = v_n\}.X_i \rightsquigarrow_c v_i \quad (\textsc{Project}) \qquad \frac{x \notin FV(v)}{(\lambda x.e)v \rightsquigarrow_c \mathsf{let\ rec}\ x = v\ \mathsf{in}\ e} \quad (\textsc{Beta})$$

$$\frac{dom(b) \perp FV(\mathbb{L})}{\mathbb{L}\,[\mathsf{let\ rec}\ b\ \mathsf{in}\ e] \rightsquigarrow_c \mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}\,[e]} \quad (\textsc{Lift})$$

- **Computational reduction rules**

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

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

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

- **Evaluation contexts**

Lift context:
$\mathbb{L} \ ::= \ \Box e \mid v\Box \mid \Box.X \mid \{\mathbb{S}\}$
Multiple lift context:
$\mathbb{F} \ ::= \ \Box \mid \mathbb{L}\,[\mathbb{F}]$
Evaluation context:
$\mathbb{E} \ ::= \ (b_v \vdash \mathbb{F}\,) \mid (\mathbb{B}\,[\mathbb{F}] \vdash e)$

Record contexts:
$\mathbb{S} \ ::= \ s_v, X = \Box, s$
Sequence contexts:
$\mathbb{B} \ ::= \ b_v, x = \Box, b$
Strict contexts:
$\mathbb{N} \ ::= \ \Box v \mid \Box.X$

- **Access in evaluation contexts**

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

Figure 7.10: Reduction semantics for $\lambda_\circ$

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* $\dashrightarrow_c$ 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 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 Lift performing the lifting operation, and a *computational reduction relation*, defined in figure 7.10.

The contraction rule 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 □ appears immediately under the first node, in position of the next sub-expression evaluated.

The second contraction rule 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

$$
\begin{array}{llll}
 & x & \in & \mathit{Vars} \\
 & X & \in & \mathit{Names} \\
\text{Expression:} & & & \\
 & E & \in & \mathit{Expr} ::= x \mid \lambda x.E \mid EE \qquad \lambda\text{-calculus} \\
 & & & \quad \mid \mathsf{let}\ x_1 = E_1 \ldots x_n = E_n\ \mathsf{in}\ E \qquad \text{Non-recursive } \mathsf{let} \text{ binding} \\
 & & & \quad \mid \{X_1 = E_1 \ldots X_n = E_n\} \mid E.X \qquad \text{Records} \\
 & & & \quad \mid l \mid \mathsf{alloc} \mid \mathsf{update} \qquad \text{Locations, allocation, mutation}
\end{array}
$$

Figure 7.11: Syntax of λ_{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 Subst. This rule transforms an expression of the shape $\mathbb{E}[\mathbb{N}[x]]$ into $\mathbb{E}[\mathbb{N}[v]]$, provided the context $\mathbb{E}[\mathbb{N}]$ defines x as v and no variable capture occurs. The meta-variable $\mathbb{N}$ ranges over *strict* contexts. A strict context is a context that requires a non-variable node to evaluate. An example of 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 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 $(b_v, x \diamond \mathbb{F}, b)$ is the topmost, partially evaluated binding, then b_v defines the variables it binds, inside $\mathbb{F}$, 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 $=_s$. Hence we can define computational reduction over equivalence classes of expressions, obtaining the reduction relation $\longrightarrow$.

7.3 The target language λ_{alloc}

The syntax of the target language λ_{alloc} is presented in figure 7.11. It distinguishes variables x from names X. It includes the constructs of the λ-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 alloc, an update operator update, and locations l.

The semantics of λ_{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 $H_v \in \mathit{HeapValues}$ is either a function $\lambda x.E$, or an evaluated record $\{S_v\}$ (where $Sv ::= X_1 = V_1 \ldots X_n = V_n$), or an application of the shape $\mathsf{alloc}\, n$, 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:

$$\begin{array}{lll} C & & ::= \Theta \vdash E \\ \Theta & \in \ \mathit{Heaps} & = \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{HeapValues} \end{array}$$

Answer:

$$\begin{array}{l} A \in \mathit{Answers} ::= \Theta \vdash V \\ V \ \in \ \mathit{Values} ::= x \mid l \end{array}$$

More meta-variables:

$$\begin{array}{ll} H_v \in \mathit{HeapValues} & ::= \lambda x.E \mid \mathsf{alloc}\ n \mid \{S_v\} \\ S_v & ::= X_1 = V_1 \ldots X_n = V_n \\ B & ::= x_1 = E_1 \ldots x_n = E_n \end{array}$$

Figure 7.12: Configurations and results in λ_{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 $\lambda x.[\![e]\!]$ in the heap Θ.

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 *Size* from λ_{alloc} heap value blocks to $\mathbb{N}$.

Notation We write $\Theta\langle l \mapsto H_v\rangle$ for the map equal to Θ anywhere but on l where it returns H_v. We write $\Theta_1 + \Theta_2$ for the union of two heaps Θ_1 and Θ_2 whose domains are disjoint. In particular, when the heap Θ is undefined on l, we write $\Theta + \{l \mapsto H_v\}$ to denote the union of Θ and $\{l \mapsto H_v\}$.

7.3.1 Structural equivalence

In λ_{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 $\mathit{Subst} = \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{Values}$. 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 $\mathit{dom}(\sigma)$ of a substitution σ 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 σ_1 and σ_2 is defined by $x\{\sigma_1 \circ \sigma_2\} = x\{\sigma_1\}\{\sigma_2\}$: it acts as σ_1, then σ_2. Moreover, we call variable renamings, or simply renamings, the injective substitutions whose codomains contain

- **Substitutions** Let $\sigma \in \mathit{Subst} = \mathit{Vars} \xrightarrow{\text{Fin}} \mathit{Values}$.

$$\begin{array}{rcl}
z\{\sigma\} & = & \sigma(z) \\
(\lambda x.E)\{\sigma\} & = & \lambda x.(E\{\sigma_{\setminus \{x\} \cup \sigma^{-1}(\{x\})}\}) \\
(\mathsf{let}\ \ B\ \ \mathsf{in}\ \ F)\{\sigma\} & = & \mathsf{let}\ \ B\{\sigma\}\ \ \mathsf{in}\ \ F\{\sigma_{\setminus\, dom(B) \cup \sigma^{-1}(dom(B))}\} \\
(x = E, B)\{\sigma\} & = & (x = E\{\sigma\}, B\{\sigma_{\setminus \{x\} \cup \sigma^{-1}(\{x\})}\}) \\
\Theta \vdash E\{\sigma\} & = & \sigma \circ \Theta \vdash E\{\sigma\} \\
 & & \text{Other cases easy.}
\end{array}$$

- **Capture**

$$\begin{array}{rcl}
\mathit{Capt}_x(\mathsf{let}\ \ y = E, B\ \ \mathsf{in}\ \ F) & = & \mathit{Capt}_x(\lambda y.\,\mathsf{let}\ \ B\ \ \mathsf{in}\ \ F) \cup \mathit{Capt}_x(E) \\
\mathit{Capt}_x(x) & = & \emptyset \\
\mathit{Capt}_x(\lambda y.E) & = & \begin{cases} \{y\} \cup \mathit{Capt}_x(E) & \\ \quad \text{if } x \in FV(\lambda y.E) & \\ \emptyset \qquad\qquad \text{otherwise} & \end{cases}
\end{array}$$

Other cases easy.

- **Unsafe new names**

$$\begin{array}{rcl}
\mathit{UnsafeNewNames}(x, \lambda x.E) & = & \mathit{Capt}_x(E) \cup FV(E) \\
\mathit{UnsafeNewNames}(x, \mathsf{let}\ \ x = E, B\ \ \mathsf{in}\ \ F) & = & \mathit{UnsafeNewNames}(x, \lambda x.(\mathsf{let}\ \ B\ \ \mathsf{in}\ \ F))
\end{array}$$

- **Structural reduction**

$$\frac{y \notin \mathit{UnsafeNewNames}(x, \lambda x.E)}{\lambda x.E \leadsto_s \lambda y.(E\{x \mapsto y\})}$$

$$\frac{y \notin \mathit{UnsafeNewNames}(x, \mathsf{let}\ \ x = E, B\ \ \mathsf{in}\ \ F)}{\mathsf{let}\ \ x = E, B\ \ \mathsf{in}\ \ F \leadsto_s \mathsf{let}\ \ y = E, (B\{x \mapsto y\})\ \ \mathsf{in}\ \ F\{x \mapsto y\}}$$

Figure 7.13: Structural equivalence in λ_{alloc}

$$\frac{\Theta(l) = \lambda x.E}{\Theta \vdash lV \rightsquigarrow_c \Theta \vdash E\{x \mapsto V\}}\ (\textsc{Beta}) \qquad \frac{l \notin dom(\Theta)}{\Theta \vdash H_v \rightsquigarrow_c \Theta + \{l \mapsto H_v\} \vdash l}\ (\textsc{Allocate})$$

$$\frac{\Theta(l) = \{S_v\}}{\Theta \vdash l.X \rightsquigarrow_c \Theta \vdash S_v(X)}\ (\textsc{Project}) \qquad \frac{Size(\Theta(l_1)) = Size(\Theta(l_2))}{\Theta \vdash \mathsf{update}\, l_1\, l_2 \rightsquigarrow_c \Theta\langle l_1 \mapsto \Theta(l_2)\rangle \vdash \{\}}\ (\textsc{Update})$$

$$\frac{dom(B) \perp \Lambda}{\Theta \vdash \Lambda[\mathsf{let}\ \ B\ \ \mathsf{in}\ \ E] \rightsquigarrow_c \Theta \vdash \mathsf{let}\ \ B\ \ \mathsf{in}\ \ \Lambda[E]}\ (\textsc{Lift})$$

Figure 7.14: Computational contraction rules for λ_{alloc}

only variables, and we denote them by ζ. Symmetrically, we call variable allocations the injective substitutions mapping variables to locations, and denote them by η.

We extend substitutions to λ_{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 is in the above example. The notion of capture is formalized by the *Capt* function in figure 7.13. Basically, $Capt_x(e)$ denotes the set of binding variables located above occurrences of x in e. For instance $Capt_x(\lambda y.x)$ 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, $\rightsquigarrow_s$, 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 $\dashrightarrow_s$ is the contextual closure of the structural contraction relation. These two relations are symmetric, and therefore the transitive closure $\dashrightarrow_s{}^*$ of $\dashrightarrow_s$ is a congruence, called the structural equivalence relation, and also written $=_s$.

7.3.2 Semantics

The semantics of λ_{alloc}, like the one of $\lambda_\circ$, 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 $\lambda_\circ$, evaluation results are values surrounded by a heap binding:

$$A \in Answers ::= \Theta \vdash V.$$

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:

$$\begin{array}{lll} \Lambda & ::= & \Box E \mid V\Box \mid \Box.X \mid \{\Sigma\} \\ & \mid & \mathsf{let}\ x = \Box, B\ \mathsf{in}\ e \end{array}$$

Record context:

$$\Sigma \ ::= \ S_v, X = \Box, S$$

Multiple lift context:

$$\Phi \ ::= \ \Box \mid \Lambda[\Phi]$$

Figure 7.15: Evaluation contexts of λ_{alloc}

$$\frac{\Theta \vdash E \rightsquigarrow_c \Theta' \vdash E'}{\Theta \vdash \Phi[E] \dashrightarrow_c \Theta' \vdash \Phi[E']} \quad (\textsc{Context})$$

$$\Theta \vdash \mathsf{let}\ x = V, B\ \mathsf{in}\ E \dashrightarrow_c \Theta \vdash (\mathsf{let}\ B\ \mathsf{in}\ E)\{x \mapsto V\} \quad (\textsc{Let})$$

$$\Theta \vdash \mathsf{let}\ \epsilon\ \mathsf{in}\ E \dashrightarrow_c \Theta \vdash E \quad (\textsc{EmptyLet}) \qquad \frac{l \notin (FV(\Theta_{\backslash\{l\}}) \cup dom(\Theta_{\backslash\{l\}}) \cup FV(E))}{\Theta \vdash E \dashrightarrow_c \Theta_{\backslash\{l\}} \vdash E} \quad (\mathrm{GC})$$

$$\Theta \vdash \mathsf{let}\ B_1\ \mathsf{in}\ \mathsf{let}\ B_2\ \mathsf{in}\ E \dashrightarrow_c \Theta \vdash \mathsf{let}\ B_1, B_2\ \mathsf{in}\ E \quad (\mathrm{EM})$$

Figure 7.16: Computational reduction in λ_{alloc}

The Project rule works similarly: it projects a name X out of a heap allocated record $l \mapsto \{S_v\}$, where S_v is a finite set of evaluated record field definitions of the shape $X_1 = V_1 \ldots X_n = V_n$. The result is $S_v(X)$ (i.e. V_i is $X = X_i$).

The Allocate rule is one of the key points of λ_{alloc}. It states that a value block H_v evaluates into a fresh heap location containing H_v, and a pointer to it: $\Theta + \{l \mapsto H_v\} \vdash l$ (l fresh). If H_v is a dummy block $\mathsf{alloc}\, n$, 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_{v1} and H_{v2} in the heap Θ, then $\Theta \vdash \mathsf{update}\, l_1\, l_2$ will evaluate to $\Theta\langle l_1 \mapsto H_{v2}\rangle \vdash \{\}$.

Finally, as in $\lambda_\circ$, 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 λ_{alloc}, lift contexts are of the shape

$$\Lambda ::= \Box E \mid V\Box \mid \Box.X \mid \{\Sigma\} \mid \mathsf{let}\ x = \Box, B\ \mathsf{in}\ e,$$

where Σ ranges over record contexts, of the shape $\Sigma ::= S_v, X = \Box, S$.

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.

7.3.3 The λ_{alloc} calculus and its confluence

The set of *terms* of the λ_{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 $\longrightarrow$.

Definition 19 *The λ_{alloc} calculus is the set of terms, equipped with the relation $\longrightarrow$.*

Unlike in $\lambda_\circ$, the reduction of λ_{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 λ_{alloc} is confluent. Let $\xrightarrow{\mathcal{C}}$ 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.

$$\begin{tikzcd}
 & & C \arrow[dll, "\mathcal{C}"'] \arrow[drr, "\text{EM}"] & & \\
C_1 \arrow[ddrr, dotted, "\text{EM}"', "*" near end] & & & & C_2 \arrow[dl, dotted, "\mathcal{C}"] \\
 & & & {} \arrow[dl, dotted, "\text{EM}", "*"' near end] & \\
 & & C' & &
\end{tikzcd}$$

Proposition 9 *For all configurations C, C_1, and C_2 such that $C \xrightarrow{\mathcal{C}} C_1$ and $C \xrightarrow{\text{EM}} C_2$, there exists a configuration C' such that $C_1 \xrightarrow{\text{EM}}{}^{*} C'$ and $C_2 \xrightarrow{\mathcal{C}}\xrightarrow{\text{EM}}{}^{*} C'$.*

Proof

If $C \xrightarrow{\textsc{EmptyLet}} C_1$, the two obtained configurations are identical. If $C \xrightarrow{\mathcal{C}} C_1$ by rule Let, then the two reductions simply commute. If $C \xrightarrow{\textsc{Context}} C_1$, then we have to examine the underlying contraction step $C \leadsto_c C_1$. 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 $E = \Lambda[\mathsf{let}\ B\ \mathsf{in}\ E_1]$, and $C_1 = \Theta \vdash \Phi[\mathsf{let}\ B\ \mathsf{in}\ \Lambda[E_1]]$.

- If $\Lambda ::= \Box F \mid V\Box \mid \{\Sigma\} \mid \Box.X$, as rule EM applies on C, we must have Φ of the shape $\mathsf{let}\ x = \Phi_1, B_1\ \mathsf{in}\ \mathsf{let}\ B_2\ \mathsf{in}\ F'$. Therefore

$$C_1 = \Theta \vdash \mathsf{let}\ x = \Phi_1[\mathsf{let}\ B\ \mathsf{in}\ \Lambda[E_1]], B_1\ \mathsf{in}\ \mathsf{let}\ B_2\ \mathsf{in}\ F',$$

and

$$C_2 = \Theta \vdash \mathsf{let}\ x = \Phi_1[\Lambda[\mathsf{let}\ B\ \mathsf{in}\ E_1]], B_1, B_2\ \mathsf{in}\ F'.$$

Let

$$C' = \Theta \vdash \mathsf{let}\ x = \Phi_1[\mathsf{let}\ B\ \mathsf{in}\ \Lambda[E_1]], B_1, B_2\ \mathsf{in}\ F'.$$

We obtain easily that C_1 and C_2 both reduce to C', in one step of $\xrightarrow{\text{EM}}$ and $\xrightarrow{\textsc{Lift}}$, respectively, which is as expected.

- If $\Lambda = \mathsf{let}\ \ x = \square, B_1\ \ \mathsf{in}\ \ F$, then Φ might still be of the shape $\mathsf{let}\, x = \Phi_1, B_1'\, \mathsf{in}\, \mathsf{let}\, B_2\, \mathsf{in}\, F'$, in which case the previous reasoning applies. If it is not of this shape, then the let binding contained in Λ is part of the EM redex, so $\Phi = \square$, and F is of the shape $\mathsf{let}\ \ B_2\ \ \mathsf{in}\ \ F''$. So, we have a diagram of the shape:

$$\begin{tikzcd}
\Theta \vdash \begin{array}{l} \mathsf{let}\, x = (\mathsf{let}\ \ B\ \ \mathsf{in}\ \ E_1), B_1 \\ \mathsf{in}\, \mathsf{let}\, B_2 \\ \mathsf{in}\, F'' \end{array} \arrow[rr, "\textsc{Lift}"] \arrow[dd, "\text{EM}"'] & & \Theta \vdash \begin{array}{l} \mathsf{let}\, B\, \mathsf{in} \\ \mathsf{let}\, x = E_1, B_1\, \mathsf{in} \\ \mathsf{let}\ \ B_2\ \ \mathsf{in}\ \ F'' \end{array} \arrow[d, "\text{EM}"] \\
 & & \Theta \vdash \begin{array}{l} \mathsf{let}\, B, x = E_1, B_1\, \mathsf{in} \\ \mathsf{let}\ \ B_2\ \ \mathsf{in}\ \ F'' \end{array} \arrow[d, "\text{EM}"] \\
\Theta \vdash \begin{array}{l} \mathsf{let}\ \ x = (\mathsf{let}\ \ B\ \ \mathsf{in}\ \ E_1), \\ \qquad B_1, B_2 \\ \mathsf{in}\, F'' \end{array} \arrow[r, "\textsc{Lift}"] & \Theta \vdash \begin{array}{l} \mathsf{let}\, B\, \mathsf{in} \\ \mathsf{let}\, x = E_1, B_1, B_2 \\ \mathsf{in}\, F'' \end{array} \arrow[r, "\text{EM}"] & \Theta \vdash \begin{array}{l} \mathsf{let}\, B, x = E_1, B_1, B_2 \\ \mathsf{in}\, F'' \end{array}
\end{tikzcd}$$

□

This result extends by a simple induction to the following corollary, pictorially described by the following diagram.

$$\begin{tikzcd}
 & C \arrow[dl, "\mathcal{C}"'] \arrow[dr, "\text{EM}^{*}"] & \\
C_1 \arrow[ddr, dotted, "\text{EM}^{*}"'] & & C_2 \arrow[d, dotted, "\mathcal{C}"] \\
 & & {} \arrow[dl, dotted, "\text{EM}^{*}"] \\
 & C' &
\end{tikzcd}$$

Corollary 7 *For all configurations C, C_1, and C_2 such that $C \xrightarrow{\mathcal{C}} C_1$ and $C \xrightarrow{\text{EM}}{}^{*} C_2$, there exists a configuration C' such that $C_1 \xrightarrow{\text{EM}}{}^{*} C'$ and $C_2 \xrightarrow{\mathcal{C}}\xrightarrow{\text{EM}}{}^{*} C'$.*

Then, the relation $\xrightarrow{\mathcal{C}\text{EM}}$ is defined as $\xrightarrow{\mathcal{C}}$, extended with rule EM. Formally, $\xrightarrow{\mathcal{C}\text{EM}} = \xrightarrow{\mathcal{C}} \cup \xrightarrow{\text{EM}}$.

Thanks to the previous corollary, we prove that the $\xrightarrow{\mathcal{C}\text{EM}}$ relation is confluent. This is done by considering the relation $\xrightarrow{\mathcal{C}\text{EM}}\xrightarrow{\text{EM}}{}^{*}$, which is strongly confluent. In other terms for any two reduction steps $C \xrightarrow{\mathcal{C}\text{EM}}\xrightarrow{\text{EM}}{}^{*} C_1$ and $C \xrightarrow{\mathcal{C}\text{EM}}\xrightarrow{\text{EM}}{}^{*} C_2$, there exist a configuration C' and two reduction steps

$C_1 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}}{}^* C'$ and $C_2 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}}{}^* C'$. A pictorial view of this is given by the following diagram:

$$\begin{tikzcd}
 & & C \arrow[dl, "\mathcal{C}\mathrm{EM}"'] \arrow[dr, "\mathcal{C}\mathrm{EM}"] & & \\
 & {} \arrow[dl, "\mathrm{EM}^*"'] & & {} \arrow[dr, "\mathrm{EM}^*"] & \\
C_1 \arrow[dr, dotted, "\mathcal{C}\mathrm{EM}"'] & & & & C_2 \arrow[dl, dotted, "\mathcal{C}\mathrm{EM}"] \\
 & {} \arrow[dr, dotted, "\mathrm{EM}^*"'] & & {} \arrow[dl, dotted, "\mathrm{EM}^*"] & \\
 & & C' & &
\end{tikzcd}$$

Proposition 10 *The relation* $\xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}}{}^*$ *is strongly confluent.*

Proof To prove this last statement, we proceed by case on the $\mathcal{C}$EM 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 $\xrightarrow{\mathcal{C}}$ steps. The only relevant case is when one reduction is a $\xrightarrow{\mathcal{C}}$ step, say $C \xrightarrow{\mathcal{C}} C_1$, and the other is in $\xrightarrow{\mathrm{EM}}$.

In this case, we have $C \xrightarrow{\mathcal{C}} C_1' \xrightarrow{\mathrm{EM}}{}^* C_1$. By the previous corollary, we obtain a C_2' such that $C_2 \xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}}{}^* C_2'$. Then, by confluence of the deterministic relation $\xrightarrow{\mathrm{EM}}$, we obtain C' such that $C_1' \xrightarrow{\mathrm{EM}}{}^* C'$ and $C_2' \xrightarrow{\mathrm{EM}}{}^* C'$. This configuration is also such that C_1 and C_2 reduce to it by relation $\xrightarrow{\mathcal{C}\mathrm{EM}}\xrightarrow{\mathrm{EM}}{}^*$, in at most one step.

This is depicted by the following diagram.

$$\begin{tikzcd}
 & & & & C \arrow[ddll, "\mathcal{C}"] \arrow[ddddrrrr, "\mathrm{EM}^*"] & & & & \\
 & & & & & & & & \\
 & & C_1' \arrow[ddll, "\mathrm{EM}^*"] \arrow[ddddrrrr, dotted, "\mathrm{EM}^*"] & & & & & & \\
 & & & & & & & & \\
C_1 \arrow[ddddrrrr, dotted, "\mathrm{EM}^*"] & & & & & & & & C_2 \arrow[dl, dotted, "\mathcal{C}"] \\
 & & & & & & & {} \arrow[dl, dotted, "\mathrm{EM}^*"] & \\
 & & & & & & C_2' \arrow[ddll, dotted, "\mathrm{EM}^*"'] & & \\
 & & & & & & & & \\
 & & & & C' & & & &
\end{tikzcd}$$

□

Corollary 8 (Confluence of λ_{alloc}) *The λ_{alloc} calclulus is confluent.*

7.4 Translation

7.4.1 Generalized contexts in λ_{alloc}

The purpose of this paper is to prove that $\lambda_\circ$ can be faithfully translated into λ_{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 $\lambda_\circ$, and could be translated as such in λ_{alloc}, but is not a result of λ_{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 λ_{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 $\lambda_\circ$ can be of the shape $b_v \vdash v$. We imagine that b_v will be translated as the heap, roughly. But heaps of λ_{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 $\mathsf{let\ rec}$ in $\lambda_\circ$. 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 $\lambda_\circ$ heaps that cannot be included in λ_{alloc} heaps as substitutions. For instance, the $\lambda_\circ$ 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 $\lambda_\circ$ context $x = y, z = x \vdash \Box$ does not correspond to any standard evaluation context in λ_{alloc}. Instead, we have to define a stronger kind of evaluation contexts, including a heap Θ, a standard context Φ, and a substitution σ. We write them $\Theta \vdash \Phi[\sigma]$, and denote them by Ψ.

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 σ is correct for an expression E iff*

$$\forall x \in dom(\sigma), \sigma(x) \notin Capt_x(E).$$

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 LET and EMPTYLET are only valid at toplevel. However, we will see that inside contexts of the shape $\Theta \vdash \Box[\sigma]$, reduction is preserved.

We first prove that contraction is preserved under correct substitution.

Proposition 11 *If $C_1 \rightsquigarrow_c C_2$ and σ is correct for C_1 and C_2, then $C_1\{\sigma\} \rightsquigarrow_c C_2\{\sigma\}$.*

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 σ is correct, $(\lambda x.E)\{\sigma\} = \lambda x.(E\{\sigma\})$. So $\Theta_1\{\sigma\} \vdash l(V\{\sigma\}) \rightsquigarrow_c \Theta_2\{\sigma\} \vdash E\{\sigma\}\{x \mapsto (V\{\sigma\})\}$. As σ is correct for C_1, x is not in the domain or codomain of σ, so $\sigma \circ \{x \mapsto (V\{\sigma\})\} = \{x \mapsto V\} \circ \sigma$, and therefore $C_1\{\sigma\} \rightsquigarrow_c \Theta_2\{\sigma\} \vdash E_2\{\sigma\}$.

Allocate, Update, Project. Similar.

Lift We again have $\Theta_1 = \Theta_2$, with $E_1 = \Lambda[\mathsf{let}\ B\ \mathsf{in}\ E]$ and $E_2 = \mathsf{let}\ B\ \mathsf{in}\ \Lambda[E]$. By the side condition on the LIFT rule, we also know that $dom(B) \perp FV(\Lambda)$. By hypothesis, we finally have $dom(B)$ disjoint from the domain and codomain of σ.

So, $C_1\{\sigma\} = \Theta_1\{\sigma\} \vdash \Lambda\{\sigma\}[\mathsf{let}\ B\{\sigma\}\ \mathsf{in}\ E\{\sigma\}]$, which reduces to $\Theta_1\{\sigma\} \vdash \mathsf{let}\, B\{\sigma\}\, \mathsf{in}\, \Lambda\{\sigma\}[E\{\sigma\}]$, as expected.

□

This property extends to computational reduction.

Proposition 12 *If $C_1 \longrightarrow C_2$ and σ 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 $C_1 = \Theta_1 \vdash \mathsf{let}\ x = V, B\ \mathsf{in}\ E$, and $C_2 = \Theta_1 \vdash \mathsf{let}\ B\{x \mapsto V\}\ \mathsf{in}\ E\{x \mapsto V\}$. So,

$$C_1\{\sigma\} = \Theta_1\{\sigma\} \vdash \mathsf{let}\ x = (V\{\sigma\}), B\{\sigma\}\ \mathsf{in}\ (E\{\sigma\}),$$

which reduces to

$$\Theta_1\{\sigma\} \vdash \mathsf{let}\ B\{\sigma\}\{x \mapsto (V\{\sigma\})\}\ \mathsf{in}\ (E\{\sigma\}\{x \mapsto (V\{\sigma\})\}),$$

but as x is not in the domain or codomain of σ, the substitution $\sigma \circ \{x \mapsto (V\{\sigma\})\}$ is equal to $\{x \mapsto V\} \circ \sigma$, so $C_1\{\sigma\}$ reduces to

$$\Theta_1\{\sigma\} \vdash \mathsf{let}\ B\{\{x \mapsto V\} \circ \sigma\}\ \mathsf{in}\ (E\{\{x \mapsto V\} \circ \sigma\}),$$

which is exactly $C_2\{\sigma\}$.

□

Now, we prove that reduction by the 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 λ_{alloc}

Proposition 13 *If* $C_1 \overset{\textsc{Context}}{\longrightarrow} C_2$*, then for any context* Ψ*,* $\Psi[C_1] \overset{\textsc{Context}}{\longrightarrow} \Psi[C_2]$.

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 σ 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 $C_1'' \overset{\textsc{Context}}{\longrightarrow} C_2''$, 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 $C_1 \longrightarrow C_2$ is of the shape:

$$\frac{\Theta_1 \vdash E_1' \rightsquigarrow_c \Theta_2 \vdash E_2'}{C_1 \longrightarrow C_2}$$

But it is trivial that contraction rules are not affected by additional bindings in the heap, so we obtain easily that

$$\Theta + \Theta_1 \vdash E_1' \rightsquigarrow_c \Theta + \Theta_2 \vdash E_2'$$

Then, by rule Context, we have

$$C_1'' \longrightarrow C_2''.$$

Finally, by proposition 12, we deduce that

$$C_1''\{\sigma\} \longrightarrow C_2''\{\sigma\},$$

which is the expected result.

□

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 ϕ, 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* $C_1 \longrightarrow C_2$*, then* $\phi[C_1] \longrightarrow \phi[C_2]$.

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:

$$\begin{array}{lcl}
\llbracket x \rrbracket & = & x \\
\llbracket \lambda x.e \rrbracket & = & \lambda x.\llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket & = & \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\
\llbracket \{ \dots X_i = e_i \dots \} \rrbracket & = & \{ \dots X_i = \llbracket e_i \rrbracket \dots \} \\
\llbracket e.X \rrbracket & = & \llbracket e \rrbracket .X \\
\llbracket \mathsf{let\ rec}\ b\ \mathsf{in}\ e \rrbracket & = & \mathsf{let}\ \mathit{Dummy}(b), \mathit{Update}(b)\ \mathsf{in}\ \llbracket e \rrbracket
\end{array}$$

Dummy pre-allocation of bindings:

$$\begin{array}{lcll}
\mathit{Dummy}(\epsilon) & = & \epsilon & \\
\mathit{Dummy}(x = e, b) & = & (x = \mathsf{alloc}\, n, \mathit{Dummy}(b)) & \text{if } \mathit{Size}(e) = n \\
\mathit{Dummy}(x = e, b) & = & \mathit{Dummy}(b) & \text{if } \mathit{Size}(e) = [?]
\end{array}$$

Computation of bindings:

$$\begin{array}{lcll}
\mathit{Update}(\epsilon) & = & \epsilon & \\
\mathit{Update}(x = e, b) & = & (y = (\mathsf{update}\, x \llbracket e \rrbracket), \mathit{Update}(b)) & \text{if } \mathit{Size}(e) = n \text{, with } y \text{ fresh} \\
\mathit{Update}(x = e, b) & = & (x = \llbracket e \rrbracket, \mathit{Update}(b)) & \text{if } \mathit{Size}(e) = [?]
\end{array}$$

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 $\lambda_\circ$ expressions to $\mathbb{N} \cup \{[?]\}$. 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 $e_\downarrow \in \mathit{Predictable}$, $\mathit{Size}(e_\downarrow) \neq [?]$, and for any variable x, $\mathit{Size}(x) = [?]$.

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 λ_{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 λ_{alloc} configurations to $\lambda_\circ$ expressions, and λ_{alloc} configurations to $\lambda_\circ$ 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. $\lambda x.\llbracket e \rrbracket$, but the result is allocated on the heap, at a fresh location l: $\{l \mapsto \lambda x.\llbracket e \rrbracket\} \vdash l$.

Translation of expressions as configurations:

$$\begin{array}{lllll}
\llbracket x \rrbracket^{\mathrm{TOP}} & = & \emptyset \vdash x & & \\
\llbracket \lambda x.e \rrbracket^{\mathrm{TOP}} & = & \{l \mapsto \lambda x.\llbracket e \rrbracket\} \vdash l & & \\
\llbracket \{s_v\} \rrbracket^{\mathrm{TOP}} & = & \Theta + \{l \mapsto \{S_v\}\} \vdash l & \text{for} & \llbracket s_v \rrbracket^{\mathrm{TOP}} = \Theta \vdash S_v \\
\llbracket \{s_v, X = e, s\} \rrbracket^{\mathrm{TOP}} & = & \Theta_1 + \Theta_2 \vdash \{S_v, X = E, \llbracket s \rrbracket\} & \text{for} & \left\{ \begin{array}{l} e \notin \mathit{values} \\ \llbracket s_v \rrbracket^{\mathrm{TOP}} = \Theta_1 \vdash S_v \\ \llbracket e \rrbracket^{\mathrm{TOP}} = \Theta_2 \vdash E \end{array} \right. \\
\llbracket ve \rrbracket^{\mathrm{TOP}} & = & \Theta_1 + \Theta_2 \vdash VE & \text{for} & \left\{ \begin{array}{l} \llbracket v \rrbracket^{\mathrm{TOP}} = \Theta_1 \vdash V \\ \llbracket e \rrbracket^{\mathrm{TOP}} = \Theta_2 \vdash E \end{array} \right. \\
\llbracket e_1 e_2 \rrbracket^{\mathrm{TOP}} & = & \Theta \vdash E\llbracket e_2 \rrbracket & \text{for} & \left\{ \begin{array}{l} e_1 \notin \mathit{values} \\ \llbracket e_1 \rrbracket^{\mathrm{TOP}} = \Theta \vdash E \end{array} \right. \\
\llbracket e.X \rrbracket^{\mathrm{TOP}} & = & \Theta \vdash E.X & \text{for} & \llbracket e \rrbracket^{\mathrm{TOP}} = \Theta \vdash E \\
\llbracket \mathsf{let\ rec}\ b\ \mathsf{in}\ e \rrbracket^{\mathrm{TOP}} & = & \left\{ \begin{array}{ll} \llbracket b \rrbracket^{\mathrm{TOP}}[\emptyset \vdash \llbracket e \rrbracket] & \text{if } b \text{ is not evaluated} \\ \llbracket b \rrbracket^{\mathrm{TOP}}[\llbracket e \rrbracket^{\mathrm{TOP}}] & \text{otherwise} \end{array} \right. & &
\end{array}$$

Translation of configurations:

$$\llbracket b \vdash e \rrbracket^{\mathrm{TOP}} = \llbracket \mathsf{let\ rec}\ b\ \mathsf{in}\ e \rrbracket^{\mathrm{TOP}}$$

Translation of bindings and evaluated records:

$$\begin{array}{lll}
\llbracket b_v, b \rrbracket^{\mathrm{TOP}} & = & \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b) \qquad \text{where } b \neq (x = v, b') \\
\llbracket X_1 = v_1 \ldots X_n = v_n \rrbracket^{\mathrm{TOP}} & = & \biguplus_{1 \leq i \leq n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n) \\
 & & \qquad\qquad\qquad \text{with } \forall i, \llbracket v_i \rrbracket^{\mathrm{TOP}} = \Theta_i \vdash V_i
\end{array}$$

Figure 7.19: The TOP translation (first part)

Translation of evaluated bindings: Ev. binding → (heap × substitution × variable allocation)

$$
\begin{array}{lcll}
\mathit{TOP}(\epsilon) & = & \emptyset \vdash (id, id) & \\
\mathit{TOP}(x = v, b_v) & = & \Theta \vdash (\sigma \circ \{x \mapsto V\}, \eta) & \text{if } \left\{ \begin{array}{l} \mathit{Size}(v) = [?] \\ [\![v]\!]^{\mathrm{TOP}} = \emptyset \vdash V \\ \mathit{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \end{array} \right. \\
\mathit{TOP}(x = v, b_v) & = & \Theta \vdash (\sigma, \eta \cup \{x \mapsto l\}) & \text{if } \left\{ \begin{array}{l} \mathit{Size}(v) = n \\ [\![v]\!]^{\mathrm{TOP}} = \Theta \vdash l \\ \mathit{TOP}(b_v) = \Theta \vdash (\sigma, \eta) \end{array} \right.
\end{array}
$$

Actual dummy pre-allocation: Binding → (heap × variable allocation)

$$
\begin{array}{lcll}
\mathit{TDum}(\epsilon) & = & \emptyset \vdash id & \\
\mathit{TDum}(x = e, b) & = & \mathit{TDum}(b) & \text{if } \mathit{Size}(v) = [?] \\
\mathit{TDum}(x = e, b) & = & \Theta + \{l \mapsto \mathsf{alloc}\, n\} \vdash \eta \cup \{x \mapsto l\} & \text{if } \left\{ \begin{array}{l} \mathit{Size}(v) = n \\ \mathit{TDum}(b) = \Theta \vdash \eta \end{array} \right.
\end{array}
$$

Actual computation of bindings: Binding → (heap × binding of λ_{alloc})

$$
\begin{array}{lcll}
\mathit{TUp}(\epsilon) & = & \emptyset \vdash \epsilon & \\
\mathit{TUp}(x = e, b) & = & \Theta_1 + \Theta_2 \vdash x = E, B & \text{if } \left\{ \begin{array}{l} \mathit{Size}(v) = [?] \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E \\ \mathit{TUp}(b) = \Theta_2 \vdash B \end{array} \right. \\
\mathit{TUp}(x = e, b) & = & \Theta_1 + \Theta_2 \vdash y = (\mathsf{update}\, x E), B & \text{if } \left\{ \begin{array}{l} \mathit{Size}(v) = n \\ [\![e]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E \\ \mathit{TUp}(b) = \Theta_2 \vdash B \\ y \text{ fresh} \end{array} \right.
\end{array}
$$

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: $\Theta + \{l \mapsto \{S_v\}\} \vdash l$. 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, $[\![v_i]\!]^{\text{TOP}} = \Theta_i \vdash V_i$, then $\Theta \vdash S_v = \biguplus_{1 \leq i \leq n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n)$.

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 Θ, 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 Θ, a substitution σ, corresponding to the definitions of unknown size, and an allocation η, giving the locations allocated in Θ for the definitions of known size. Semantically, it corresponds to a heap Θ 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 Θ and an allocation η, and it forms a context $\Theta \vdash \square[\eta]$. The *TUp* function returns a heap Θ and a binding B, which form a context $\Theta \vdash \mathsf{let}\ B\ \mathsf{in}\ \square[id]$. The *TOP* function returns a heap Θ, a substitution σ, and an allocation η, 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, $\mathit{TDum}(b)$ and $\mathit{TUp}(b)$ are empty, and the translation of $\mathsf{let\ rec}\ b_v, b\ \mathsf{in}\ e$ is the TOP translation of e, $[\![e]\!]^{\mathrm{TOP}}$, put in the context $\mathit{TOP}(b_v)$. Otherwise, the translation of $\mathsf{let\ rec}\ b_v, b\ \mathsf{in}\ e$ is the standard translation of e, put in the context $\mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b)$.

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 σ, for all context $\mathbb{C}$:*

- *If $\mathit{Size}(e) = n$ and $b \vdash e \longrightarrow b' \vdash e'$, then $\mathit{Size}(e') = n$;*
- *If $\mathit{Size}(v) = n$, then there exist Θ and l such that $[\![v]\!]^{\mathrm{TOP}} = \Theta \vdash l$ and $\mathit{Size}(\Theta(l)) = n$;*
- *If $\mathit{Size}(e) = \mathit{Size}(f) = n$, then $\mathit{Size}(\mathbb{C}[e]) = \mathit{Size}(\mathbb{C}[f])$.*
- *$\mathit{Size}(e\{\sigma\}) = \mathit{Size}(e)$;*
- *$\mathit{Size}(\mathsf{let\ rec}\ b\ \mathsf{in}\ e) = \mathit{Size}(e)$.*

Proposition 15 (Translation of values reduces to TOP) *For all context Ψ and for all value v, $\Psi[\emptyset \vdash [\![v]\!]] \longrightarrow^* \Psi[[\![v]\!]^{\mathrm{TOP}}]$, 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.
- $v = \{X_1 = v_1 \dots X_n = v_n\}$. By induction hypothesis, for any context Ψ_i, for each i, we have

  $$\Psi_i[\emptyset \vdash [\![v_i]\!]] \longrightarrow \Psi_i[[\![v_i]\!]^{\mathrm{TOP}}].$$

  Let for each i, $[\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i$. By a trivial induction on n, we prove that for any context Ψ,

  $$\Psi[\emptyset \vdash [\![\{X_1 = v_1 \dots X_n = v_n\}]\!]] \longrightarrow^* \Psi[\biguplus_{1 \leq i \leq n} \Theta_i \vdash \{X_1 = V_1 \dots X_n = V_n\}],$$

  only by rule Context (Allocate). By proposition 13, this configuration in turn reduces by rule Context (Allocate) to

  $$\Psi[\biguplus_{1 \leq i \leq n} \Theta_i + \{l \mapsto \{X_1 = V_1 \dots X_n = V_n\}\} \vdash l],$$

  which is exactly $[\![v]\!]^{\mathrm{TOP}}$.

□

Corollary 9 *For all weak evaluation context ϕ, expression E, and binding b of the shape $b = (x = v, b')$,*

$$\phi \circ \mathit{Update}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \mathit{TUp}(b)[\emptyset \vdash E]$$

Proof We know that $\phi \circ \mathit{Update}(b)[\emptyset \vdash E] = \phi[\mathsf{let}\ \ y = \Phi[[\![v]\!]], \mathit{Update}(b')\ \ \mathsf{in}\ \ E]$, where $(y, \Phi) = \begin{cases} (x, \Box) \text{ if } \mathit{Size}(v) = [?] \\ (z, \mathsf{update}\, x\ \Box) \text{ otherwise } (z \text{ fresh}). \end{cases}$

This expression can be seen as $\Psi[\emptyset \vdash v]$ for some Ψ. By proposition 15, it reduces to $\Psi[[\![v]\!]^{\mathrm{TOP}}]$, so we obtain $\phi \circ \mathsf{let}\ \ y = \Phi, \mathit{Update}(b')\ \ \mathsf{in}\ \ E[[\![v]\!]^{\mathrm{TOP}}]$, which is exactly $\phi \circ \mathit{TUp}(b)[\emptyset \vdash E]$. □

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 *TOP* function, which is defined only on evaluated bindings, whereas the second part is given as an argument to the *TDum* and *TUp* 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. $\mathit{TDum}({b_v}', b') \circ \mathit{TOP}(b_v) \circ \mathit{TUp}({b_v}', b')$. 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 *Dummy* function, and at the meta level by the *TDum* function, in the same way ; then we prove a similar property for the functions *Update* and *TUp* ; and we eventually connect the two to prove the whole desired property.

Proposition 16 (Dummy) *For all binding B, for all weak evaluation context ϕ,*

$$\phi[\emptyset \vdash \mathsf{let}\ \ \mathit{Dummy}(b), B\ \ \mathsf{in}\ \ E] \longrightarrow^* (\phi \circ \mathit{TDum}(b))[\emptyset \vdash \mathsf{let}\ \ B\ \ \mathsf{in}\ \ E].$$

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 $\mathit{Size}(e) = [?]$. Then $\mathit{Dummy}(b) = \mathit{Dummy}(b')$ and $\mathit{TDum}(b) = \mathit{TDum}(b')$, so by induction hypothesis, we obtain the expected result.
- $b = (x = e, b')$, with $\mathit{Size}(e) = n$. Then $\mathit{Dummy}(b) = (x = \mathsf{alloc}\, n, \mathit{Dummy}(b'))$. Let $\mathit{TDum}(b') = \Theta \vdash \eta$, we have $\mathit{TDum}(b) = \Theta + \{l \mapsto \mathsf{alloc}\, n\} \vdash \eta \cup \{x \mapsto l\}$, for a fresh l. Let ϕ be a weak evaluation context, and $E_0 = \phi[\mathsf{let}\ \ \mathit{Dummy}(b), B\ \ \mathsf{in}\ \ E]$. We have $E_0 = \phi[\emptyset \vdash \mathsf{let}\ \ x = \mathsf{alloc}\, n, \mathit{Dummy}(b'), B\ \ \mathsf{in}\ \ E]$. By rule Context (Allocate), we have $E_0 \longrightarrow \phi[\{l \mapsto \mathsf{alloc}\, n\} \vdash \mathsf{let}\ \ x = l, \mathit{Dummy}(b'), B\ \ \mathsf{in}\ \ E]$. By proposition 14, this last expression reduces to $\phi[\{l \mapsto \mathsf{alloc}\, n\} \vdash (\mathsf{let}\ \ \mathit{Dummy}(b'), B\ \ \mathsf{in}\ \ E)\{x \mapsto l\}]$. Let $\phi_0 = \mathit{TDum}(x = e) = \{l \mapsto \mathsf{alloc}\, n\} \vdash \Box[\{x \mapsto l\}]$ and $\phi_1 = \phi \circ \phi_0$; we can view the expression as $\phi_1[\emptyset \vdash \mathsf{let}\ \ \mathit{Dummy}(b'), B\ \ \mathsf{in}\ \ E]$, which by induction hypothesis reduces to $\phi_1[\mathit{TDum}(b')[\emptyset \vdash \mathsf{let}\ \ B\ \ \mathsf{in}\ \ E]]$. In other words, we obtain $\phi[\mathit{TDum}(x = e) \circ \mathit{TDum}(b')[\emptyset \vdash \mathsf{let}\ \ B\ \ \mathsf{in}\ \ E]]$, which is the expected result, since obviously $\mathit{TDum}(x = e) \circ \mathit{TDum}(b') = \mathit{TDum}(b)$.

□

Proposition 17 (Update) *Let $b = (x = v, b')$. For all weak evaluation context ϕ, for all expression E, we have*

$$\phi \circ \mathit{TDum}(b) \circ \mathit{TUp}(b)[\emptyset \vdash E] \longrightarrow^* \phi \circ \mathit{TDum}(b') \circ \mathit{TOP}(x = v) \circ \mathit{Update}(b')[\emptyset \vdash E].$$

Proof

- If $Size(v) = n$, then $\llbracket v \rrbracket^{\mathrm{TOP}} = \Theta_v \vdash l$, and we have

  $$TUp(b) = \Theta_v \vdash y = \mathsf{update}\, x\ l, Update(b'),$$

  with a fresh y. Alternatively, we can choose another fresh location l' for the result, and have $\llbracket v \rrbracket^{\mathrm{TOP}} = \Theta'_v \vdash l'$, with $\Theta'_v = \Theta_{v\setminus l} + \{l' \mapsto \Theta_v(l)\}$.

  Let $E_0 = \phi \circ TDum(b) \circ TUp(b)[\emptyset \vdash E]$.

  We have $E_0 = \phi \circ TDum(b)[\Theta'_v \vdash \mathsf{let}\ \ y = \mathsf{update}\, x\ l', Update(b')\ \ \mathsf{in}\ \ E]$, and also $Size(v) = n$ and $TDum(b) = TDum(b') \circ (\{l \mapsto \mathsf{alloc}\, n\} \vdash \{x \mapsto l\})$. So

  $$E_0 = \phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \mathsf{alloc}\, n\} \vdash \mathsf{let}\ \ y = \mathsf{update}\, x\ l', Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto l\}].$$

  But by hypothesis 3, $Size(\Theta'_v(l')) = n$, so rule UPDATE applies, and E_0 reduces to

  $$\phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\} \vdash \mathsf{let}\ \ y = \{\}, Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto l\}],$$

  and then, as y is fresh, by rule LET to

  $$\phi \circ TDum(b')[(\Theta'_v + \{l \mapsto \Theta'_v(l')\} \vdash \mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto l\}].$$

  But the location l' is not used anymore, so by rule GC, the obtained expression reduces to

  $$\phi \circ TDum(b')[(\Theta'_{v\setminus l'} + \{l \mapsto \Theta'_v(l')\} \vdash \mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto l\}].$$

  And finally, we notice that $\Theta'_{v\setminus l'} + \{l \mapsto \Theta'_v(l')\} = \Theta_v$, so E_0 reduces to

  $$\begin{aligned} &\phi \circ TDum(b')[(\Theta_v \vdash \mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto l\}] \\ &= \phi \circ TDum(b') \circ TOP(x = v)[\emptyset \vdash \mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E] \\ &= \phi \circ TDum(b') \circ TOP(x = v) \circ Update(b')[\emptyset \vdash E]. \end{aligned}$$

- If $Size(v) = [?]$, then there exists a y such that $\llbracket v \rrbracket^{\mathrm{TOP}} = \emptyset \vdash y$, so

  $$TUp(b) = \emptyset \vdash x = y, Update(b').$$

  Let $E_0 = \phi \circ TDum(b) \circ TUp(b)[\emptyset \vdash E]$.

  We have $E_0 = \phi \circ TDum(b)[\emptyset \vdash \mathsf{let}\ \ x = y, Update(b')\ \ \mathsf{in}\ \ E]$,

  and by rule LET, by proposition 14, $E_0 \longrightarrow \phi \circ TDum(b)[\emptyset \vdash (\mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E)\{x \mapsto y\}]$.

  But $TOP(x = v) = TOP(x = y) = \emptyset \vdash (x \mapsto y, id)$, so $E_0 \longrightarrow \phi \circ TDum(b) \circ TOP(x = v)[\emptyset \vdash \mathsf{let}\ \ Update(b')\ \ \mathsf{in}\ \ E]$, which is the expected result.

□

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 σ as a context $\emptyset \vdash \square[\sigma]$.

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*

$$TOP(b_v) = \eta_{{b_v}'} \circ TOP(x = v) \circ TOP({b_v}').$$

Proof Let $TOP(x = v) = \Theta_v \vdash (\sigma_v, \eta_v)$, and $TOP(b_v') = \Theta \vdash (\sigma, \eta)$. We have $TOP(b_v) = \Theta_v + \Theta_{b_v'} \vdash (\sigma \circ \sigma_v, \eta \cup \eta_v)$. By proposition 18, we can choose Θ, σ, and η such that $\eta = \eta_{b_v'}$. Then,

$$\begin{aligned}&\eta_{b_v'} \circ TOP(x = v) \circ TOP(b_v')\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta \circ \sigma_v \circ \eta_v \circ \eta_{b_v'}\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta_{b_v'} \circ \sigma_v \circ \eta_v \circ \eta_{b_v'}\end{aligned}$$

But η_v and $\eta_{b_v'}$ have disjoint domains and codomains, so they commute and we obtain

$$\begin{aligned}&\eta_{b_v'} \circ TOP(x = v) \circ TOP(b_v')\\ &= \Theta + \Theta_v \vdash \sigma \circ \eta_{b_v'} \circ \sigma_v \circ \eta_{b_v'} \circ \eta_v\end{aligned}$$

Furthermore, $\eta_{b_v'}$ and σ_v also have disjoint domains and codomains, so they commute. Finally, $\eta_{b_v'}$ is idempotent, so

$$\begin{aligned}&\eta_{b_v'} \circ TOP(x = v) \circ TOP(b_v')\\ &= \Theta + \Theta_v \vdash \sigma \circ \sigma_v \circ \eta_{b_v'} \circ \eta_v\\ &= \Theta + \Theta_v \vdash (\sigma \circ \sigma_v) \circ (\eta_{b_v'} \cup \eta_v)\\ &= TOP(b_v)\end{aligned}$$

□

Proposition 20 (Commuting contexts) *Let $\phi_1 = \Theta_1 \vdash \square[\sigma_1]$ and $\phi_2 = \Theta_2 \vdash \square[\sigma_2]$. If $dom(\sigma_2) \perp \sigma_1$ 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 $dom(\sigma_2) \perp \sigma_1$. 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 $x \in dom(\sigma_2)$, by hypothesis $x \notin dom(\sigma_1)$, so we have $\sigma'(x) = x\{\sigma_1\}\{\sigma_2\}\{\sigma_1\} = x\{\sigma_2\}\{\sigma_1\} = \sigma(x)$.
- On $x \notin dom(\sigma_2)$, distinguish the two cases.
  - If $x \in dom(\sigma_1)$, then $\sigma(x) = x\{\sigma_2\}\{\sigma_1\} = x\{\sigma_1\}$. But by hypothesis $\sigma_1(x) \in cod(\sigma_1) \perp dom(\sigma_2)$, 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 $x \notin dom(\sigma_1)$, then $\sigma(x) = x = \sigma'(x)$.

□

Corollary 10 *Let $b_v = (b_{v1}, b_{v2})$ be a syntactically correct binding. Let $TDum(b_{v2}) = \Theta_2 \vdash \eta_2$. We have $\eta_2 \circ TOP(b_{v1}) \circ \eta_2 = \eta_2 \circ TOP(b_{v1})$.*

Proof Let $TOP(b_{v1}) = \Theta_1 \vdash (\sigma_1, \eta_1)$. By proposition 20, it is enough to prove $dom(\sigma_1 \circ \eta_1) \perp \eta_2$ and $\eta_2^2 = \eta_2$. But we have $dom(b_{v1}) \perp dom(b_{v2})$, so $dom(\sigma_1 \circ \eta_1) \perp dom(\eta_2)$. Moreover, $cod(\eta_2)$ contains only locations, whereas $dom(\sigma_1 \circ \eta_1)$ contains only variables, so $cod(\sigma_2) \perp dom(\sigma_1 \circ \eta_1)$. Finally, as all variable allocations, η_2 is idempotent. □

Corollary 11 *Let $b_v = (b_{v1}, b_{v2})$ and $TDum(b_{v2}) = \Theta_2 \vdash \eta_2$. We have*

$$TOP(b_v) = \eta_2 \circ TOP(b_{v1}) \circ TOP(b_{v2}).$$

Proof By induction on b_{v1}.

- $b_{v1} = \epsilon$, because η_2 is idempotent.

- $b_{v1} = (x = v, b_{v1}')$. Let $b_v{}' = b_{v1}', b_{v2}$, $\mathit{TDum}(b_v{}') = \Theta'_{b_v{}'} \vdash \eta_{b_v{}'}$, and $\mathit{TDum}(b_{v1}') = \Theta'_{b_{v1}'} \vdash \eta_{b_{v1}'}$. By definition of *TDum*, we have $\eta_{b_v{}'} = \eta_{b_{v1}'} \cup \eta_2$. Then, we can calculate

$$
\begin{aligned}
\mathit{TOP}(b_v) &= \eta_{b_v{}'} \circ \mathit{TOP}(x = v) \circ \mathit{TOP}(b_v{}') \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by lemma 19)} \\
&= \eta_{b_v{}'} \circ \mathit{TOP}(x = v) \circ \eta_2 \circ \mathit{TOP}(b_{v1}') \circ \mathit{TOP}(b_{v2}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction hypothesis)} \\
&= \eta_{b_{v1}'} \cup \eta_2 \circ \mathit{TOP}(x = v) \circ \eta_2 \circ \mathit{TOP}(b_{v1}') \circ \mathit{TOP}(b_{v2}) \\
&= \eta_{b_{v1}'} \circ \underline{\eta_2 \circ \mathit{TOP}(x = v) \circ \eta_2} \circ \mathit{TOP}(b_{v1}') \circ \mathit{TOP}(b_{v2}) \\
&= \eta_{b_{v1}'} \circ \eta_2 \circ \mathit{TOP}(x = v) \circ \mathit{TOP}(b_{v1}') \circ \mathit{TOP}(b_{v2}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by proposition 10)} \\
&= \eta_2 \circ \underline{\eta_{b_{v1}'} \circ \mathit{TOP}(x = v) \circ \mathit{TOP}(b_{v1}')} \circ \mathit{TOP}(b_{v2}) \\
&= \eta_2 \circ \mathit{TOP}(b_{v1}) \circ \mathit{TOP}(b_{v2}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by proposition 19)}
\end{aligned}
$$

□

Proposition 21 (TOP Update pass) *For all weak evaluation context ϕ, and configuration C,*

$$\phi \circ \mathit{TDum}(b_v, b) \circ \mathit{TUp}(b_v, b)[C] \longrightarrow^* \phi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[C].$$

Proof By induction on b_v. If $b_v = \epsilon$, there is nothing to prove. Otherwise, let $b_v = (x = v, b_v{}')$. By proposition 17,

$$\phi \circ \mathit{TDum}(b_v, b) \circ \mathit{TUp}(b_v, b)[C] \longrightarrow^* \phi \circ \mathit{TDum}(b_v{}', b) \circ \mathit{TOP}(x = v) \circ \mathit{Update}(b_v{}', b)[C].$$

But by corollary 10, this is equal to

$$\phi \circ \eta \circ \mathit{TOP}(x = v) \circ \mathit{TDum}(b_v{}', b) \circ \mathit{Update}(b_v{}', b)[C],$$

where $\mathit{TDum}(b_v{}', b) = \Theta \vdash \eta$.

By induction hypothesis, we know that the obtained expression reduces to

$$\phi \circ \eta \circ \mathit{TOP}(x = v) \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v{}') \circ \mathit{Update}(b)[C].$$

But if we let $\mathit{TDum}(b_v{}') = \Theta_1 \vdash \eta_1$ and $\mathit{TDum}(b) = \Theta_2 \vdash \eta_2$, we have $\eta = \eta_1 \cup \eta_2$, so

$$
\begin{aligned}
&\phi \circ \eta \circ \mathit{TOP}(x = v) \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v{}') \circ \mathit{Update}(b)[C] \\
&= \phi \circ \eta_1 \circ \underline{\eta_2 \circ \mathit{TOP}(x = v) \circ \mathit{TDum}(b)} \circ \mathit{TOP}(b_v{}') \circ \mathit{Update}(b)[C] \\
&= \phi \circ \eta_1 \circ \mathit{TDum}(b) \circ \mathit{TOP}(x = v) \circ \mathit{TOP}(b_v{}') \circ \mathit{Update}(b)[C] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by corollary 10)} \\
&= \phi \circ \mathit{TDum}(b) \circ \underline{\eta_1 \circ \mathit{TOP}(x = v) \circ \mathit{TOP}(b_v{}')} \circ \mathit{Update}(b)[C] \\
&\qquad\qquad \text{(because } \mathit{TDum}(b) \text{ is not modified by any substitution)} \\
&\phi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[C] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by proposition 19)}
\end{aligned}
$$

□

Proposition 22 (Update pass) *For all weak evaluation context ϕ, and configuration C,*

$$\phi \circ \mathit{TDum}(b_v, b) \circ \mathit{Update}(b_v, b)[C] \longrightarrow^* \phi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[C].$$

Proof By corollary 9, we have

$$
\begin{aligned}
&\phi \circ \mathit{TDum}(b_v, b) \circ \mathit{Update}(b_v, b)[C] \\
&\longrightarrow^* \phi \circ \mathit{TDum}(b_v, b) \circ \mathit{TUp}(b_v, b)[C].
\end{aligned}
$$

By proposition 21, it further reduces to $\phi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[C]$. □

Proposition 23 (Partial translation of bindings) *For all evaluation context Ψ,*

$$\Psi[\emptyset \vdash [\![\mathsf{let\ rec}\ \ b_v, b\ \ \mathsf{in}\ \ e]\!]] \longrightarrow^* \Psi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[\emptyset \vdash e].$$

Proof Let $\Psi = \Theta \vdash \Phi[\sigma]$, and $\phi = \Theta \vdash \Box[\sigma]$. Let

$$E_0 = \Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash \mathsf{let}\ \ \mathit{Dummy}(b_v, b), \mathit{Update}(b_v, b)\ \ \mathsf{in}\ \ [\![e]\!]]$$

By rule Lift and modulo variable renaming, we have

$$E_0 \longrightarrow^* \phi[\emptyset \vdash \mathsf{let}\ \ \mathit{Dummy}(b_v, b), \mathit{Update}(b_v, b)\ \ \mathsf{in}\ \ \Phi[[\![e]\!]]].$$

By proposition 16, this expression reduces to $\phi \circ \mathit{TDum}(b_v, b)[\emptyset \vdash \mathsf{let}\ \ \mathit{Update}(b_v, b)\ \ \mathsf{in}\ \ \Phi[[\![e]\!]]]$.

By proposition 22, it in turn reduces to $\phi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[\Phi[[\![e]\!]]]$, which is equal to $\Psi \circ \mathit{TDum}(b) \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b)[[\![e]\!]]$. □

Lemma 39 (Standard translation reduces to TOP translation) *For all context Ψ and for all expression e,*

$$\Psi[\emptyset \vdash [\![e]\!]] \longrightarrow^* \Psi[[\![e]\!]^{\mathrm{TOP}}].$$

Proof By induction on e. If e is a value, we use proposition 15.

Application. Let $e = e_1 e_2$, Ψ be a context, and $E_0 = \Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash [\![e_1]\!][\![e_2]\!]]$. Let also $[\![e_1]\!]^{\mathrm{TOP}} = \Theta_1 \vdash E_1$. By induction hypothesis, $E_0 \longrightarrow^* \Psi[\Theta_1 \vdash E_1[\![e_2]\!]]$. If e_1 is not a value, this is directly $\Psi[[\![e]\!]^{\mathrm{TOP}}]$. Otherwise, E_1 is a value, say V_1, and $\Psi_0 = \Psi[\Theta_1 \vdash (V_1 \Box)[\mathit{id}]]$ is an evaluation context, so by induction hypothesis again, if we let $[\![e_2]\!]^{\mathrm{TOP}} = \Theta_2 \vdash E_2$, then $\Psi_0[\emptyset \vdash [\![e_2]\!]] \longrightarrow^* \Psi_0[\Theta_2 \vdash E_2]$, which is equal to $\Psi[\Theta_1 + \Theta_2 \vdash V_1 E_2] = \Psi[[\![e]\!]^{\mathrm{TOP}}]$.

Record field selection. Simple by induction hypothesis.

Record. Let $e = \{s_v, X = f, s\}$, where f is not a value. Let $[\![s_v]\!]^{\mathrm{TOP}} = \Theta_1 \vdash S_v$. 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 $[\![f]\!]^{\mathrm{TOP}} = \Theta_2 \vdash F$. 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 $e = \mathsf{let\ rec}\ \ b\ \ \mathsf{in}\ \ f$.

1. If $b = \epsilon$, then $[\![b]\!]^{\mathrm{TOP}} = \emptyset \vdash \Box[\mathit{id}]$, so $[\![e]\!]^{\mathrm{TOP}} = [\![f]\!]^{\mathrm{TOP}}$. So, $\Psi[\emptyset \vdash [\![e]\!]] = \Psi[\emptyset \vdash \mathsf{let}\ \ \epsilon\ \ \mathsf{in}\ \ [\![f]\!]]$. By rules Lift and then EmptyLet, it reduces to $\Psi[\emptyset \vdash [\![f]\!]]$, which by induction hypothesis reduces to $\Psi[[\![f]\!]^{\mathrm{TOP}}]$, as expected.
2. If $b = b_v$, non empty, then $[\![e]\!]^{\mathrm{TOP}} = \mathit{TOP}(b_v)[[\![f]\!]^{\mathrm{TOP}}]$. We have

$$\begin{array}{l} \Psi[\emptyset \vdash [\![e]\!]] \\ = \Psi[\emptyset \vdash [\![\mathsf{let\ rec}\ \ b_v\ \ \mathsf{in}\ \ f]\!]] \\ \longrightarrow^* \Psi \circ \mathit{TOP}(b_v)[\emptyset \vdash [\![f]\!]] \\ \qquad\qquad \text{(by proposition 23)} \\ \longrightarrow^* \Psi \circ \mathit{TOP}(b_v)[[\![f]\!]^{\mathrm{TOP}}] \\ \quad \text{(by induction hypothesis)} \\ = \Psi[[\![e]\!]^{\mathrm{TOP}}] \end{array}$$

3. If $b = b_v, b'$, with b' non empty, then $\llbracket e \rrbracket^{\mathrm{TOP}} = \mathit{TDum}(b') \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b')[\emptyset \vdash \llbracket f \rrbracket]$. We have

$$\begin{aligned}
&\Psi[\emptyset \vdash \llbracket e \rrbracket] \\
&= \Psi[\emptyset \vdash \llbracket \mathsf{let\ rec}\ \ b_v, b'\ \ \mathsf{in}\ \ f \rrbracket] \\
&\longrightarrow^{*} \Psi \circ \mathit{TDum}(b') \circ \mathit{TOP}(b_v) \circ \mathit{Update}(b')[\emptyset \vdash \llbracket f \rrbracket] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by proposition 23)} \\
&\longrightarrow^{*} \Psi \circ \mathit{TDum}(b') \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b')[\emptyset \vdash \llbracket f \rrbracket] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction hypothesis)} \\
&= \Psi[\llbracket e \rrbracket^{\mathrm{TOP}}]
\end{aligned}$$

□

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 $y = \mathsf{update}\, x\ \llbracket e \rrbracket$, whereas otherwise, it is translated as $x = \llbracket e \rrbracket$. For this reason, it is not compositional in the usual sense: a straightforward property such as $\llbracket \mathbb{E}\,[e] \rrbracket = \llbracket \mathbb{E} \rrbracket[\llbracket e \rrbracket]$ does not hold. Moreover, there is no straightforward translation for contexts: consider $\mathsf{let\ rec}\ \ x = \square\ \ \mathsf{in}\ \ \{\}$ 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 $\lambda_\circ$, 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 $\llbracket \square_\zeta \rrbracket^{\mathrm{TOP}} = \llbracket \square_\zeta \rrbracket = \square$, given that a context hole $\square_\zeta$ has size ζ, 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 $\llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_\zeta$ for $\llbracket \mathbb{E}\,[\square_\zeta] \rrbracket^{\mathrm{TOP}}$. The standard translation is compositional for this notion of contexts.

Proposition 24 (Compositionality of the standard translation) *For all context $\mathbb{E}$ and expression e,*

$$\llbracket \mathbb{E}\,[e] \rrbracket = \llbracket \mathbb{E} \rrbracket_{\mathit{Size}(e)}[\llbracket e \rrbracket].$$

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 $e \notin$ Values, then*

$$\llbracket \mathbb{L}\,[e] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(e)}[\llbracket e \rrbracket^{\mathrm{TOP}}].$$

Proof By case analysis on $\mathbb{L}$. We treat one example case, application: $\mathbb{L} = \square f$. We have $\llbracket \mathbb{L}\,[e] \rrbracket^{\mathrm{TOP}} = \llbracket e f \rrbracket^{\mathrm{TOP}} = \Theta \vdash E\llbracket f \rrbracket$, where $\llbracket e \rrbracket^{\mathrm{TOP}} = \Theta \vdash E$. But $\llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(e)} = \emptyset \vdash \square\llbracket f \rrbracket$, which is the expected result. □

Proposition 26 (Compositionality for multiple lift contexts) *If $e \notin$ Values, then*

$$\llbracket \mathbb{F}\,[e] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(e)}[\llbracket e \rrbracket^{\mathrm{TOP}}].$$

Proof By induction on $\mathbb{F}$. If $\mathbb{F} = \Box$, there is nothing to prove. Otherwise, let $\mathbb{F} = \mathbb{L}\,[\mathbb{F}']$ and $\zeta = \mathit{Size}(e)$.

By induction hypothesis, $\llbracket \mathbb{F}'[e] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}]$.

As the *Size* function is compositional, $\zeta' = \mathit{Size}(\mathbb{F}'[e]) = \mathit{Size}(\mathbb{F}'[\Box_\zeta])$.

By proposition 26, $\llbracket \mathbb{L}\,[\mathbb{F}'[e]] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}'[e] \rrbracket^{\mathrm{TOP}}] = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}]]$.

By proposition 26, $\llbracket \mathbb{L}\,[\mathbb{F}'] \rrbracket^{\mathrm{TOP}}_{\zeta} = \llbracket \mathbb{L}\,[\mathbb{F}'[\Box_\zeta]] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}'[\Box_\zeta] \rrbracket^{\mathrm{TOP}}] = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}]$.

So, $\llbracket \mathbb{L}\,[\mathbb{F}'[e]] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{L}\,[\mathbb{F}'] \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}]$. □

Lemma 40 (Compositionality for evaluation contexts) *If $e \notin \mathit{Values}$, then*

$$\llbracket \mathbb{E}\,[e] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(e)}[\llbracket e \rrbracket^{\mathrm{TOP}}].$$

Proof By case on $\mathbb{E}$. Let $\zeta = \mathit{Size}(e)$.

- If $\mathbb{E} = \mathbb{F}$, use proposition 26.
- If $\mathbb{E} = b_v \vdash \mathbb{F}$, then
$$\begin{aligned}\llbracket \mathbb{E}\,[e] \rrbracket^{\mathrm{TOP}} &= \mathit{TOP}(b_v)[\llbracket \mathbb{F}\,[e] \rrbracket^{\mathrm{TOP}}] \\ &= \mathit{TOP}(b_v)[\llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}]] \\ &= (\mathit{TOP}(b_v) \circ \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta})[\llbracket e \rrbracket^{\mathrm{TOP}}] \\ &= \llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}].\end{aligned}$$
- If $\mathbb{E} = (b_v, x = \mathbb{F}, b \vdash f)$, then let $b_0 = (x = \mathbb{F}\,[e], b)$. We have $\llbracket \mathbb{E}\,[e] \rrbracket^{\mathrm{TOP}} = \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b_0)[\emptyset \vdash \llbracket f \rrbracket]$, since $\mathbb{F}\,[e]$ cannot be a value.

  Let $\Theta' \vdash E' = \llbracket \mathbb{F}\,[e] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}]$ (by proposition 26).

  Let $\Theta_u \vdash B = \mathit{TUp}(b)$, and $\zeta' = \mathit{Size}(\mathbb{F}\,[e]) = \mathit{Size}(\mathbb{F}\,[\Box_\zeta])$.

  Let $(x', \Phi') = \left\{ \begin{array}{l} (x, E') \text{ if } \zeta' = [?] \\ (x, \mathsf{update}\, x\ E') \text{ otherwise} \end{array} \right.$

  We have $\mathit{TUp}(b_0) = \Theta_u + \Theta' \vdash x' = \Phi'[E'], B$. Let $\Psi_0 = \Theta_u \vdash \mathsf{let}\ x' = \Phi', B\ \mathsf{in}\ \llbracket f \rrbracket$. We have
$$\begin{aligned}\llbracket \mathbb{E}\,[e] \rrbracket^{\mathrm{TOP}} &= \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \Psi_0 \circ \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}] \\ &= \llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket e \rrbracket^{\mathrm{TOP}}].\end{aligned}$$

□

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 Ψ,*

$$\Psi[\llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(v)}[\llbracket v \rrbracket^{\mathrm{TOP}}]] \longrightarrow^* \Psi[\llbracket \mathbb{L}\,[v] \rrbracket^{\mathrm{TOP}}].$$

Proof By case on $\mathbb{L}$. Let $\zeta = \mathit{Size}(v)$ and $\Theta_v \vdash V = \llbracket v \rrbracket^{\mathrm{TOP}}$.

- If $\mathbb{L}$ is of the shape $v'\Box$ or $\Box.X$, then $\Psi[\llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(v)}[\llbracket v \rrbracket^{\mathrm{TOP}}]] = \Psi[\llbracket \mathbb{L}\,[v] \rrbracket^{\mathrm{TOP}}]$.
- $\mathbb{L} = \Box e$. Let $\llbracket e \rrbracket^{\mathrm{TOP}} = \Theta \vdash E$. We have $\llbracket e \rrbracket^{\mathrm{TOP}}_{\zeta} = \emptyset \vdash \Box\llbracket e \rrbracket$ and $\Psi \circ \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] = \Psi[\Theta_v \vdash V\llbracket e \rrbracket]$, which by lemma 39 reduces to $\Psi[\Theta_v + \Theta \vdash VE] = \Psi[\llbracket \mathbb{L}\,[v] \rrbracket^{\mathrm{TOP}}]$.

- $\mathbb{L} = \{s_v, X = \square, s\}$. Let $\llbracket s_v \rrbracket^{\mathrm{TOP}} = \Theta'_v \vdash {S_v}'$, $\llbracket s \rrbracket = S$, and $\llbracket s \rrbracket^{\mathrm{TOP}} = \Theta' \vdash S'$.

  We have $\Psi \circ \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] = \Psi[\Theta_v + \Theta'_v \vdash \{{S_v}', X = V, S\}]$, which by lemma 39 reduces to $\Psi[C] = \Psi[\Theta_v + \Theta'_v + \Theta' \vdash \{{S_v}', X = V, S'\}]$. If s is not evaluated, then C is exactly $\llbracket \mathbb{L}\,[v] \rrbracket^{\mathrm{TOP}}$. Otherwise, $\Psi[C]$ reduces by rule Context (Allocate) to $\Psi[\Theta_v + \Theta'_v + \Theta' + \{l \mapsto \{{S_v}', X = V, S'\}\} \vdash l]$, which is exactly $\Psi[\llbracket \mathbb{L}\,[v] \rrbracket^{\mathrm{TOP}}]$.

□

Proposition 28 (Semi-compositionality for multiple lift contexts) *For all evaluation context Ψ,*

$$\Psi[\llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(v)}[\llbracket v \rrbracket^{\mathrm{TOP}}]] \longrightarrow^{*} \Psi[\llbracket \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}}].$$

Proof By induction on $\mathbb{F}$. If $\mathbb{F} = \square$, there is nothing to prove. Otherwise, $\mathbb{F} = \mathbb{L}\,[\mathbb{F}']$. Let $\zeta = \mathit{Size}(v)$ and $\zeta' = \mathit{Size}(\mathbb{F}'[\square_{\zeta}]) = \mathit{Size}(\mathbb{F}'[v])$ (by hypothesis 3).

By proposition 27, as neither $\mathbb{F}'[\square_{\zeta}]$ nor $\mathbb{F}'[v]$ are values, we have $\llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta} = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}]$ and $\llbracket \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}} = \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}'[v] \rrbracket^{\mathrm{TOP}}]$.

By induction hypothesis,

$$\begin{array}{l}
\Psi \circ \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
= \Psi \circ \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'} \circ \llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
= \Psi \circ \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}' \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}]] \\
\longrightarrow^{*} \Psi \circ \llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}'[v] \rrbracket^{\mathrm{TOP}}] \\
= \Psi[\llbracket \mathbb{L} \rrbracket^{\mathrm{TOP}}_{\zeta'}[\llbracket \mathbb{F}'[v] \rrbracket^{\mathrm{TOP}}]] \\
= \Psi[\llbracket \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}}]
\end{array}$$

□

Proposition 29 (Semi-compositionality for evaluation contexts) *For all evaluation context Ψ,*

$$\Psi[\llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\mathit{Size}(v)}[\llbracket v \rrbracket^{\mathrm{TOP}}]] \longrightarrow^{*} \Psi[\llbracket \mathbb{E}\,[v] \rrbracket^{\mathrm{TOP}}].$$

Proof By case analysis on $\mathbb{E}$.

- $\mathbb{E} = (b_v \vdash \mathbb{F})$. Let $\zeta = \mathit{Size}(v)$ and $\zeta' = \mathit{Size}(\mathbb{F}\,[\square_{\zeta}]) = \mathit{Size}(\mathbb{F}\,[v])$ (by hypothesis 3). We have

$$\begin{array}{l}
(\Psi \circ \llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\zeta})[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
= \Psi \circ \mathit{TOP}(b_v) \circ \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
\longrightarrow^{*} \Psi \circ \mathit{TOP}(b_v)[\llbracket \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}}] \\
\qquad\qquad \text{(by proposition 28)} \\
= \Psi[\llbracket b_v \vdash \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}}] \\
= \Psi[\llbracket \mathbb{E}\,[v] \rrbracket^{\mathrm{TOP}}].
\end{array}$$

- $\mathbb{E} = (\mathbb{B}\,[\mathbb{F}] \vdash e)$, with $\mathbb{B} = (b_v, x = \square, b)$. Let $\zeta = \mathit{Size}(v)$ and $\zeta' = \mathit{Size}(\mathbb{F}\,[\square_{\zeta}]) = \mathit{Size}(\mathbb{F}\,[v])$ (by hypothesis 3). Let also $b_0 = (x = \square_{\zeta'}, b)$. We have

$$\begin{array}{l}
\Psi \circ \llbracket \mathbb{E} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
= \Psi \circ \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ (\mathit{TUp}(b_0)[\emptyset \vdash \llbracket e \rrbracket]) \circ \llbracket \mathbb{F} \rrbracket^{\mathrm{TOP}}_{\zeta}[\llbracket v \rrbracket^{\mathrm{TOP}}] \\
\longrightarrow^{*} \Psi \circ \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ (\mathit{TUp}(b_0)[\emptyset \vdash \llbracket e \rrbracket])[\llbracket \mathbb{F}\,[v] \rrbracket^{\mathrm{TOP}}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{(by proposition 28)}
\end{array}$$

  If $\mathbb{F}\,[v]$ is not a value, the obtained expression is exactly $\Psi[\llbracket \mathbb{E}\,[v] \rrbracket^{\mathrm{TOP}}]$. Otherwise, the obtained expression is a partial translation of $\mathbb{E}\,[v]$, so by proposition 23, it reduces to $\Psi[\llbracket \mathbb{E}\,[v] \rrbracket^{\mathrm{TOP}}]$, as expected.

□

7.5.2 Translation of access

In $\lambda_\circ$, 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 λ_{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* $\mathit{TOP}(b_v) = \Theta_a \vdash (\sigma, \eta)$*,* $b_v(x) = v$*, and* $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash V$*, then* $\Theta_v \subset \Theta_a$ *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 $\mathit{Size}(v) = n$. We have

$$\begin{array}{rcl} [\![v]\!]^{\mathrm{TOP}} & = & \Theta_v \vdash l \\ \mathit{TOP}(b_v{}') & = & \Theta'_a \vdash \sigma'\eta' \\ \mathit{TOP}(b_v) & = & Th_v + \Theta'_a \vdash (\sigma', (\eta' + \{x \mapsto l\})) = \Theta_a \vdash (\sigma, \eta) \end{array}$$

  Obviously, we have $\Theta_v \subset \Theta_a$. Furthermore, by syntactic correctness of b_v, $x \notin \mathit{dom}(\sigma)$, so $(\sigma \circ \eta)(x) = \eta(x) = l = V = V\{\sigma \circ \eta\}$.

- $b_v = (x = v, b_v{}')$, with $\mathit{Size}(v) = [?]$. We have

$$\begin{array}{rcl} [\![v]\!]^{\mathrm{TOP}} & = & \emptyset \vdash y = \Theta_v \vdash V \\ \mathit{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\ \mathit{TOP}(b_v) & = & \Theta'_a \vdash (\sigma' \circ \{x \mapsto y\}, \eta'), \end{array}$$

  and therefore $(\sigma \circ \eta)(x) = y\{\eta'\} = V\{\eta\}$.

- $b_v = (y = v', b_v{}')$ and $\mathit{Size}(v') = n$. We have

$$\begin{array}{rcl} [\![v']\!]^{\mathrm{TOP}} & = & \Theta'_v \vdash l \\ \mathit{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\ \mathit{TOP}(b_v) & = & \Theta'_a + \Theta'_v \vdash (\sigma', \eta' + \{y \mapsto l\}) = \Theta_a \vdash (\sigma, \eta). \end{array}$$

  By induction hypothesis, $\Theta_v \subset \Theta'_a$, so $\Theta_v \subset \Theta'_a$. 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 $\mathit{Size}(v') = \mathit{undefined}$. We have

$$\begin{array}{rcl} [\![v']\!]^{\mathrm{TOP}} & = & \emptyset \vdash z \\ \mathit{TOP}(b_v{}') & = & \Theta'_a \vdash (\sigma', \eta') \\ \mathit{TOP}(b_v) & = & \Theta'_a \vdash (\sigma' \circ \{y \mapsto z\}, \eta') = \Theta_a \vdash (\sigma, \eta). \end{array}$$

  By induction hypothesis, $\Theta_v \subset \Theta'_a$, so $\Theta_v \subset \Theta'_a$. 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 $y \notin \mathit{cod}(\eta')$, and as we additionally have $y \notin \mathit{dom}(\eta')$, we can deduce that $\{y \mapsto z\} \circ \eta' = \eta' \circ \{y \mapsto z\{\eta'\}\}$. So, we have

$$\begin{array}{l} (\sigma \circ \eta)(x) \\ = x\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\ = x\{\sigma' \circ \eta' \circ \{y \mapsto z\{\eta'\}\}\} \\ = ((\sigma' \circ \eta')(x))\{y \mapsto z\{\eta'\}\} \\ = V\{\sigma' \circ \eta'\}\{y \mapsto z\{\eta'\}\} \\ = V\{\sigma' \circ \eta' \circ \{y \mapsto z\{\eta'\}\}\} \\ = V\{\sigma' \circ \{y \mapsto z\} \circ \eta'\} \\ = V\{\sigma \circ \eta\}. \end{array}$$

□

Proposition 31 (Access) *Let* $\Psi = [\![\mathbb{E}]\!]^{\text{TOP}}_{\zeta} = \Theta \vdash \Phi[\sigma]$. *If* $\mathbb{E}\,(x) = v$ *and* $[\![v]\!]^{\text{TOP}} = \Theta_v \vdash V$, *then* $\sigma(x) = V\{\sigma\}$ *and* $\Theta_v \subset \Theta$.

Proof By case analysis on the proof of $\mathbb{E}\,(x) = v$.

EA. $\mathbb{E} = b_v \vdash \mathbb{F}$, and $b_v(x) = v$. We have

$$[\![\mathbb{E}]\!]^{\text{TOP}}_{\zeta} = \mathit{TOP}(b_v) \circ [\![\mathbb{F}]\!]^{\text{TOP}}_{\zeta}.$$

Let $\mathit{TOP}(b_v) = \Theta_a \vdash (\sigma_a, \eta_a)$ and $[\![\mathbb{F}]\!]^{\text{TOP}}_{\zeta} = \Theta' \vdash \Phi'[\mathit{id}]$. We can deduce $\sigma = \sigma_a \circ \eta_a$. By proposition 30, we have $\Theta_v \subset \Theta_a \subset \Theta$ and $(\sigma_a \circ \eta_a)(x) = V\{\sigma_a \circ \eta_a\}$, or in other words $\sigma(x) = V\{\sigma\}$, which is the expected result.

IA. $\mathbb{E} = (b_v, x = \mathbb{F}, b \vdash e)$. Then, $[\![\mathbb{E}]\!]^{\text{TOP}}_{\zeta} = \mathit{TDum}(x = \mathbb{F}[\square_\zeta], b) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(x = \mathbb{F}[\square_\zeta], b)[\emptyset \vdash [\![e]\!]]$.

$$\begin{array}{lll} \text{Let } \mathit{TDum}(x = \mathbb{F}[\square_\zeta], b) & = & \Theta_d \vdash \eta_d \\ \quad\ \mathit{TOP}(b_v) & = & \Theta_a \vdash (\sigma_a, \eta_a) \\ \quad\ \mathit{TUp}(x = \mathbb{F}[\square_\zeta], b)[\emptyset \vdash [\![e]\!]] & = & \Theta' \vdash \Phi'. \end{array}$$

We have $\sigma = \sigma_a \circ \eta_a \circ \eta_d$. By proposition 30, $\Theta_v \subset \Theta_a$, so $\Theta_v \subset \Theta$. Furthermore, $(\sigma_a \circ \eta_a)(x) = V\{\sigma_a \circ \eta_a\}$, so $\sigma(x) = x\{\sigma_a \circ \eta_a \circ \eta_d\} = x\{\sigma_a \circ \eta_a\}\{\eta_d\} = V\{\sigma_a \circ \eta_a\}\{\eta_d\} = V\{\sigma\}$, as expected.

□

7.5.3 Translation of internal merging

Proposition 32 (Internal merging) *If* $b \vdash e \xrightarrow{\text{IM}} b' \vdash e'$, *then* $[\![b \vdash e]\!]^{\text{TOP}} \longrightarrow^* [\![b' \vdash e']\!]^{\text{TOP}}$.

Proof Let $b \vdash e = (b_v, x = (\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ e_1), b_1 \vdash f)$, and $b' \vdash e' = (b_v, b_1, x = e_1, b_2 \vdash f)$. Let $b_0 = (x = (\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ e_1), b_2)$ and $b'_0 = (x = e_1, b_2)$.

We have $[\![b \vdash e]\!]^{\text{TOP}} = \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b_0)[\emptyset \vdash [\![f]\!]]$.

Let now $(x', \Phi') = \left\{ \begin{array}{l} (x, \square) \text{ if } \mathit{Size}(e_1) = \mathit{Size}(\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ e_1) = [?] \text{ (cf hypothesis 3)} \\ (y, \mathsf{update}\, x\ \square) \text{ with } y \text{ fresh otherwise.} \end{array} \right.$

Let also $\Theta_1 \vdash E_1$ be defined as follows. If b_1 is evaluated, let $\Theta_1 \vdash E_1 = [\![e_1]\!]^{\text{TOP}}$, and otherwise $\Theta_1 \vdash E_1 = \emptyset \vdash [\![e_1]\!]$. This way, we always have $[\![\mathsf{let\ rec}\ \ b_1\ \ \mathsf{in}\ \ e_1]\!]^{\text{TOP}} = [\![b_1]\!]^{\text{TOP}}[\Theta_1 \vdash E_1]$.

Finally, let $\Phi_1 = \emptyset \vdash \mathsf{let}\ \ x' = \Phi', \mathit{Update}(b_2)\ \ \mathsf{in}\ \ [\![f]\!]$, and $b_1 = b_{v1}, b'_1$, where b'_1 does not begin with a value. We have

$$\begin{array}{l} \mathit{TUp}(b_0)[\emptyset \vdash f] \\ = \Phi_1[[\![b_1]\!]^{\text{TOP}}[\Theta_1 \vdash E_1]] \\ = \Phi_1 \circ \mathit{TDum}(b'_1) \circ \mathit{TOP}(b_{v1}) \circ \mathit{TUp}(b'_1)[\Theta_1 \vdash E_1]. \end{array}$$

But the context $\mathit{TDum}(b'_1) \circ \mathit{TOP}(b_{v1})$ 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 Φ_1, and

$$\begin{array}{l} \mathit{TUp}(b_0)[\emptyset \vdash f] \\ = \mathit{TDum}(b'_1) \circ \mathit{TOP}(b_{v1}) \circ \Phi_1 \circ \mathit{TUp}(b'_1)[\Theta_1 \vdash E_1]. \end{array}$$

Now, if b_1 is not fully evaluated, the two translation are semantically identical. But if b_1 is fully evaluated, i.e. $b'_1 = \epsilon$, then $[\![b' \vdash e']\!]^{\text{TOP}}$ 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 $\mathit{TUp}(b_1') = \Theta_1' \vdash B_1'$. We have $\Theta_1 \vdash E_1 = \emptyset \vdash \llbracket e_1 \rrbracket$ and with $\phi = \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \mathit{TDum}(b_1') \circ \mathit{TOP}(b_{v1})$,

   $$\begin{array}{l} \llbracket b \vdash e \rrbracket^{\mathrm{TOP}} \\ = \phi[\Theta_1' \vdash \mathsf{let}\ x' = \mathsf{let}\ B_1'\ \mathsf{in}\ \llbracket e_1 \rrbracket [,]\, \mathit{Update}(b_2)\ \mathsf{in}\ \llbracket f \rrbracket] \\ \xrightarrow{\textsc{Lift}} \phi[\Theta_1' \vdash \mathsf{let}\ B_1'\ \mathsf{in}\ \mathsf{let}\ x' = \llbracket e_1 \rrbracket [,]\, \mathit{Update}(b_2)\ \mathsf{in}\ \llbracket f \rrbracket] \\ \xrightarrow{\mathrm{EM}} \phi[\Theta_1' \vdash \mathsf{let}\ B_1', x' = \llbracket e_1 \rrbracket [,]\, \mathit{Update}(b_2)\ \mathsf{in}\ \llbracket f \rrbracket] \\ = \phi[\Theta_1' \vdash \mathsf{let}\ B_1', x' = \llbracket e_1 \rrbracket [,]\, \mathit{Update}(b_2)\ \mathsf{in}\ \llbracket f \rrbracket] \\ = \phi \circ \mathit{TUp}(b_1', b_0')[\emptyset \vdash \llbracket f \rrbracket]. \end{array}$$

   But let us now examine ϕ a bit $\mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \mathit{TDum}(b_1') \circ \mathit{TOP}(b_{v1})$. First, notice that $\mathit{TDum}(b_0) = \mathit{TDum}(b_0')$, by hypothesis 3.

   Then, $\mathit{TOP}(b_v)$ and $\mathit{TDum}(b_1')$ are two weak evaluation contexts, and the domain of the substitution of $\mathit{TDum}(b_1')$ is included in $\mathit{dom}(b_1')$, which is disjoint from the free variables of b_v, so if $\mathit{TDum}(b_1') = \Theta_d' \vdash \eta_d'$, then $\mathit{TOP}(b_v) \circ \mathit{TDum}(b_1') = \eta_d' \circ \mathit{TOP}(b_v) \circ \mathit{TDum}(b_1')$. Moreover, η_d' is a variable allocation, and is therefore idempotent, so we can apply proposition 20 to obtain

   $$\begin{array}{rcl} \phi & = & \mathit{TDum}(b_0') \circ \mathit{TDum}(b_1') \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}) \\ & = & \mathit{TDum}(b_0', b_1') \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}) \\ & = & \mathit{TDum}(b_1', b_0') \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}). \end{array}$$

   Furthermore, $\mathit{TOP}(b_{v1}) = \Theta_{b_{v1}} \vdash (\sigma_{b_{v1}}, \eta_{b_{v1}})$. As $\eta_{b_{v1}}$ is idempotent, we have $\mathit{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathit{TOP}(b_{v1})$. But we know that the domain of $\eta_{b_{v1}}$ is disjoint from the free variables of $\mathit{TOP}(b_v)$, so $\mathit{TOP}(b_v) \circ \eta_{b_{v1}} = \eta_{b_{v1}} \circ \mathit{TOP}(b_v)$, and therefore $\phi = \mathit{TDum}(b_1', b_0') \circ \eta_{b_{v1}} \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1})$. But by corollary 10, $\eta_{b_{v1}} \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}) = \mathit{TOP}(b_v, b_{v1})$, so $\phi = \mathit{TDum}(b_1', b_0') \circ \mathit{TOP}(b_v, b_{v1})$.

   Finally, we obtain that

   $$\begin{array}{rcl} \llbracket b \vdash e \rrbracket^{\mathrm{TOP}} & = & \mathit{TDum}(b_1', b_0') \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TUp}(b_1', b_0')[\emptyset \vdash \llbracket f \rrbracket] \\ & = & \llbracket b_v, b_{v1}, b_1', b_0' \rrbracket^{\mathrm{TOP}}[\emptyset \vdash \llbracket f \rrbracket] \\ & = & \llbracket b_v, b_1, x = e_1, b_2 \rrbracket^{\mathrm{TOP}}[\emptyset \vdash \llbracket f \rrbracket] \\ & = & \llbracket b' \vdash e' \rrbracket^{\mathrm{TOP}}. \end{array}$$

2. b_1 is fully evaluated. We have $\llbracket b \vdash e \rrbracket^{\mathrm{TOP}} = \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}) \circ \Phi_1[\Theta_1 \vdash E_1]$.

   Let $\mathit{TOP}(b_{v1}) = \Theta_{b_{v1}} \vdash (\sigma_{b_{v1}}, \eta_{b_{v1}})$. We know that $\eta_{b_{v1}}$ is idempotent, so $\mathit{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathit{TOP}(b_{v1})$. As above, $\mathit{dom}(\eta_{b_{v1}}) \perp \mathit{FV}(\mathit{TOP}(b_v))$, so $\mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1}) = \eta_{b_{v1}} \circ \mathit{TOP}(b_v) \circ \mathit{TOP}(b_{v1})$, in which by corollary 10 we recognize $\mathit{TOP}(b_v, b_{v1})$.

   Therefore, $\llbracket b \vdash e \rrbracket^{\mathrm{TOP}} = \mathit{TDum}(b_0) \circ \mathit{TOP}(b_v, b_{v1}) \circ \Phi_1[\Theta_1 \vdash E_1]$.

   But we notice that $\Phi_1[\Theta_1 \vdash E_1] = \mathit{TUp}(b_0')[\emptyset \vdash \llbracket f \rrbracket]$. And by hypothesis 3, $\mathit{TDum}(b_0) = \mathit{TDum}(b_0')$. Let $\mathit{TDum}(b_0) = \Theta_{b_0} \vdash \eta_{b_0}$. By proposition 20, we have $\mathit{TDum}(b_0) \circ \mathit{TOP}(b_v, b_{v1}) = \eta_{b_0} \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TDum}(b_0')$, so $\llbracket b \vdash e \rrbracket^{\mathrm{TOP}} = \eta_{b_0} \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TDum}(b_0') \circ \mathit{TUp}(b_0')[\emptyset \vdash \llbracket f \rrbracket]$.

   Let $b_0' = (b_{v0}, b_0'')$, with b_0'' not beginning with a value. By proposition 21, $\llbracket b \vdash e \rrbracket^{\mathrm{TOP}} \longrightarrow^* \eta_{b_0} \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TDum}(b_0'') \circ \mathit{TOP}(b_{v0}) \circ \mathit{Update}(b_0'')[\emptyset \vdash \llbracket f \rrbracket]$.

   But if $\mathit{TDum}(b_{v0}) = \Theta_{b_{v0}} \vdash \eta_{b_{v0}}$ and $\mathit{TDum}(b_0'') = \Theta_{b_0''} \vdash \eta_{b_0''}$, then $\eta_{b_0} = \eta_{b_{v0}} + \eta_{b_0''}$, so by proposition 20, the obtained expression is equal to $\eta_{b_{v0}} \circ \mathit{TDum}(b_0'') \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TOP}(b_{v0}) \circ \mathit{Update}(b_0'')[\emptyset \vdash \llbracket f \rrbracket]$. But $\eta_{b_{v0}}$ commutes with $\mathit{TDum}(b_0'')$, so we obtain $\mathit{TDum}(b_0'') \circ \eta_{b_{v0}} \circ \mathit{TOP}(b_v, b_{v1}) \circ \mathit{TOP}(b_{v0}) \circ \mathit{Update}(b_0'')[\emptyset \vdash \llbracket f \rrbracket]$, which by corollary 10 is equal to $\mathit{TDum}(b_0'') \circ \eta_{b_{v0}} \circ \mathit{TOP}(b_v, b_{v1}, b_{v0}) \circ \mathit{Update}(b_0'')[\emptyset \vdash \llbracket f \rrbracket]$, which is exactly $\llbracket b' \vdash e' \rrbracket^{\mathrm{TOP}}$.

□

• **Evaluated binding contexts**
$\mathbb{B}_v ::= b_{v1}, x = \Box, b_{v2}$ with $Depth(b_{v1}, x = \Box, b_{v2})$ defined as $1+ \mid b_{v1} \mid$
• **Depth of an evaluation context**

$$\begin{array}{rcl}
Depth(\Box) & = & 0 \\
Depth(\mathbb{L}\,[\mathbb{F}\,]) & = & 1 + Depth(\mathbb{F}\,) \\
Depth(b_v \vdash \mathbb{F}\,) & = & 1 + \mid b_v \mid + Depth(\mathbb{F}\,) \\
Depth(\mathbb{B}_v\,[\mathbb{F}\,] \vdash e) & = & Depth(\mathbb{B}_v\,) + Depth(\mathbb{F}\,)
\end{array}$$

• **Measuring the number of** let rec **nodes**
$\mu_{\mathbf{l}}(e)$ is the number of let rec nodes not under a λ in e (same for configurations).
• **Measuring the depth of the** let rec **to lift** (same for configurations)

$$\begin{array}{rcl}
\mu_{\mathbf{d}}(\mathbb{F}\,[\mathbb{L}\,[\textsf{let rec}\ b_v\ \textsf{in}\ e]]) & = & 1 + Depth(\mathbb{F}\,) \\
\mu_{\mathbf{d}}(e) & = & 0 \text{ otherwise}
\end{array}$$

well defined since the sum of the depths of let rec nodes strictly decreases.
• **Measuring the binding level of the hot variable**
$\mu_{\mathbf{b}}(e)$ is the depth of the binder of the hot variable, if any:

$$\begin{array}{rcll}
\mu_{\mathbf{b}}(\mathbb{B}_v\,[v], y = \mathbb{F}\,[x], b \vdash e) & = & Depth(\mathbb{B}_v\,) & \text{if } (\mathbb{B}_v\,[v])(x) = v \\
\mu_{\mathbf{b}}(\mathbb{B}_v\,[v] \vdash \mathbb{F}\,[x]) & = & Depth(\mathbb{B}_v\,) & \text{if } (\mathbb{B}_v\,[v])(x) = v \\
\mu_{\mathbf{b}}(e) & = & 0 & \text{otherwise}
\end{array}$$

• **Measure** $\mu_{\mathbf{e}}(e) = (\mu_{\mathbf{l}}(e), \mu_{\mathbf{d}}(e))$ (lexicographically ordered).
$\mu(c) = (\mu_{\mathbf{l}}(c), \mu_{\mathbf{d}}(c), \mu_{\mathbf{b}}(c))$

Figure 7.21: Measure

7.5.4 Simulation

Due to their different ways of handling bindings, the two calculus $\lambda_\circ$ and λ_{alloc} do not yield a step by step simulation. Indeed, a redex and its reduct in $\lambda_\circ$ may have the same translation. As an example, consider any expressions of the shape $\mathbb{L}\,[\textsf{let rec}\ b_v\ \textsf{in}\ e]$ and $\textsf{let rec}\ b_v\ \textsf{in}\ \mathbb{L}\,[e]$. The binding b_v is translated as a heap Θ and a substitution σ, 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 $\lambda_\circ$ could be translated as an empty one in λ_{alloc}, thus possibly changing the infinite looping observable behaviour. 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, $\mu_{\mathbf{l}}$, is the number of let rec nodes not under a lambda in the given expression. The second, $\mu_{\mathbf{d}}$ is the depth of the let rec node to lift in the given expression, if any. Formally, if e is of the shape $\mathbb{F}\,[\mathbb{L}\,[\textsf{let rec}\ b\ \textsf{in}\ f]]$, then the let rec node can be lifted by rule Lift, so the result is the depth of the context $\mathbb{F}\,[\mathbb{L}\,]$, or 1 plus the depth of $\mathbb{F}$.

The functions $\mu_{\mathbf{l}}$ and $\mu_{\mathbf{d}}$ form a measure $\mu_{\mathbf{e}}$ on expressions, defined by $\mu_{\mathbf{e}} = (\mu_{\mathbf{l}}, \mu_{\mathbf{d}})$, ordered lexicographically.

Moreover, these two functions are straightforwardly extended to configurations, replacing $\mathbb{F}$ with $\mathbb{E}$ for the second definition.

A third function $\mu_{\mathbf{b}}$ 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 $\mu_{\mathbf{b}}(e)$ is the depth at which x is bound in $\mathbb{E}$. Formally, we define evaluated binding contexts as binding contexts of the shape $b_{v1}, x = \Box, b_{v2}$, and their depth as 1 plus the cardinal of b_{v1}. 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 $\mu_{\mathbf{e}}(e) > \mu_{\mathbf{e}}(e')$, then for any evaluation context $\mathbb{E}$, $\mu(\mathbb{E}[e]) > \mu(\mathbb{E}[e'])$.*

Proof The property clearly holds for both measures $\mu_{\mathbf{l}}$ and $\mu_{\mathbf{d}}$, thus for their lexicographic product as well. □

Lemma 41 (Contraction simulated) *If $e \leadsto_c e'$, then $[\![e]\!]^{\mathrm{TOP}} \longrightarrow^+ [\![e']\!]^{\mathrm{TOP}}$ or $[\![e]\!]^{\mathrm{TOP}} = [\![e']\!]^{\mathrm{TOP}}$ and for any $\mathbb{E}$, $\mu(\mathbb{E}[e]) > \mu(\mathbb{E}[e'])$.*

Proof By case analysis on the applied rule.

Beta. $e = ((\lambda x.f)v)$, and $e' = \mathit{letrecinx = vf}$. Let $[\![v]\!]^{\mathrm{TOP}} = \Theta_v \vdash V$. We have $[\![e]\!]^{\mathrm{TOP}} = \Theta_v + \{l \mapsto (\lambda x.[\![f]\!])\} \vdash lV$, which reduces by rule Beta to $\Theta_v + \{l \mapsto (\lambda x.[\![f]\!])\} \vdash f\{x \mapsto V\}$.

Let us now calculate $\mathit{TOP}(x = v)$.

- If $\mathit{Size}(v) = [?]$, then $\Theta_v \vdash V = \emptyset \vdash V$, and $\mathit{TOP}(x = v) = \emptyset \vdash (x \mapsto V, \mathit{id})$; so $[\![x = v]\!]^{\mathrm{TOP}} = \emptyset \vdash \Box[x \mapsto V] = \Theta_v \vdash \Box[x \mapsto V]$.
- Otherwise, $\Theta_v \vdash V = \Theta_v \vdash l$, and $\mathit{TOP}(x = v) = \Theta_v \vdash (\mathit{id}, x \mapsto l)$; so $[\![x = v]\!]^{\mathrm{TOP}} = \Theta_v \vdash \Box[x \mapsto l] = \Theta_v \vdash \Box[x \mapsto V]$.

So, in both cases, we have $[\![x = v]\!]^{\mathrm{TOP}} = \Theta_v \vdash \Box[x \mapsto V]$. Therefore, $[\![x = v]\!]^{\mathrm{TOP}}$ reduces to $[\![x = v]\!]^{\mathrm{TOP}}[[\![f]\!]]$, which by lemma 39 reduces to $[\![x = v]\!]^{\mathrm{TOP}}[[\![f]\!]^{\mathrm{TOP}}]$, which is exactly $[\![e']\!]^{\mathrm{TOP}}$.

Project. $e = \{s_v\}.X$ and $e' = s_v(X)$. Let $s_v = (X_1 = v_1 \ldots X_n = v_n)$, $X = X_{i_0}$, and for each i, $[\![v_i]\!]^{\mathrm{TOP}} = \Theta_i \vdash V_i$. We have $[\![s_v]\!]^{\mathrm{TOP}} = \biguplus_{1 \le i \le n} \Theta_i \vdash (X_1 = V_1 \ldots X_n = V_n)$, and $[\![e]\!]^{\mathrm{TOP}} = \biguplus_{1 \le i \le n} \Theta_i + \{l \mapsto \{X_1 = V_1 \ldots X_n = V_n\}\} \vdash l.X$. By rule Project, it reduces to $\biguplus_{1 \le i \le n} \Theta_i + \{l \mapsto \{X_1 = V_1 \ldots X_n = V_n\}\} \vdash V_{i_0}$, which by rule GC reduces to $\Theta_{i_0} \vdash V_{i_0}$, which is exactly $[\![e']\!]^{\mathrm{TOP}}$.

Lift. $e = \mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ f]$ and $e' = \mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[f]$.

- If b is evaluated, then $[\![e]\!]^{\mathrm{TOP}} = [\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathit{TOP}(b)[[\![f]\!]^{\mathrm{TOP}}]$. Let $\mathit{TOP}(b) = \Theta \vdash (\Box, \sigma)$. In the context $[\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathit{TOP}(b)$, the only substitution is σ, whose domain is $\mathit{dom}(b)$, which by the side condition to the Lift rule is disjoint from the free variables of $\mathbb{L}$, so the contexts commute, and $[\![e]\!]^{\mathrm{TOP}} = \mathit{TOP}(b) \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}[[\![f]\!]^{\mathrm{TOP}}] = [\![e']\!]^{\mathrm{TOP}}$.
- If b is not evaluated, then $b = b_v, b'$, with b' non empty and not beginning with a value. We have $[\![e]\!]^{\mathrm{TOP}} = [\![\mathbb{L}]\!]^{\mathrm{TOP}} \circ \mathit{TDum}(b') \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b')[\emptyset \vdash [\![f]\!]]$. But as above, the context $[\![\mathbb{L}]\!]^{\mathrm{TOP}}$ has not substitution and is not affected by the ones of $\mathit{TDum}(b')$, $\mathit{TOP}(b_v)$, and $\mathit{TUp}(b')$. So $[\![e]\!]^{\mathrm{TOP}} = \mathit{TDum}(b') \circ \mathit{TOP}(b_v) \circ \mathit{TUp}(b') \circ [\![\mathbb{L}]\!]^{\mathrm{TOP}}[\emptyset \vdash [\![f]\!]] = [\![e']\!]^{\mathrm{TOP}}$.

This is the only case where the two translations are directly equal. We thus have to show that $\mu_{\mathbf{d}}(e) > \mu_{\mathbf{d}}(e')$. And indeed $\mu_{\mathbf{d}}(e) = \mu_{\mathbf{d}}(\mathbb{L}[\mathsf{let\ rec}\ b\ \mathsf{in}\ f]) = 2 + 0$, whereas $\mu_{\mathbf{d}}(e') = \mu_{\mathbf{d}}(\mathsf{let\ rec}\ b\ \mathsf{in}\ \mathbb{L}[f]) = 0$. 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 $c = (x = \{X = \lambda y.y\} \vdash (x.X)x)$. It reduces by rule Subst to $c' = (x = \{X = \lambda y.y\} \vdash (\{X = \lambda y.y\}.X)x)$. By the 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 Θ' of C' contains an additional copy of the record *and* the function. This phenomenon happens at each application of the Subst rule. But except in case of a faulty configuration (see below), such a reduction step is necessarily followed by a Beta or a Project step. In our example, a 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 Beta reduction step. In both cases, the translated configuration reduces in one step, by the same rule (Project or 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 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 Project rule, it remains, as shown by the translation of c'':

$$C'' = \left\{ \begin{array}{l} l_1 \mapsto \lambda y.y, \\ l_2 \mapsto \{X = l_1\}, \\ l_3 \mapsto \lambda y.y \end{array} \right\} \vdash l_3 l_2.$$

Our solution to this problem consists in only considering expressions where all the record fields are variables, which we call $\mathbf{R}$-normal expressions. Any expression can be transformed into an $\mathbf{R}$-normal one, by applying the following NameFields rule, in any context.

$$\frac{\exists i, e_i \notin \mathit{Vars} \qquad \forall i, j, x_i \notin \mathit{FV}(e_j)}{\{X_1 = e_1 \ldots X_n = e_n\} \xrightarrow{\mathbf{R}} \mathsf{let\ rec}\ x_1 = e_1 \ldots x_n = e_n\ \mathsf{in}\ \{X_1 = x_1 \ldots X_n = x_n\}} \quad (\textsc{NameFields})$$

This process necessarily terminates since the number of records not containing only variables stricly decreases. The reduction rules of $\lambda_\circ$ obviously preserve the $\mathbf{R}$-normality. This way, after a sequence of a Subst step followed by a 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 $\lambda_\circ$ configuration is said *stuck on a free variable* when it is of the shape $\mathbb{E}[\mathbb{N}[x]]$ and $\mathbb{E}(x)$ is undefined. This definition is extended to λ_{alloc} configurations (replace $\mathbb{E}$ with Ψ). 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',

- 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 $\lambda_\circ$ 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 $\boldsymbol{R}$-normal configuration c, if $c \longrightarrow c'$ and $[\![c]\!]^{\mathrm{TOP}} = C$, then one of the four situations below holds:*

1. *Either c' is faulty, and then C is faulty too ;*

$$\begin{tikzcd}
c \arrow[r] \arrow[d, "{[\![]\!]}"'] & c' \arrow[r, "/" marking] & {} \\
C \arrow[r, "/" marking] & {} &
\end{tikzcd}$$

2. *or there exists C' such that $[\![e']\!] = C'$ and $C \longrightarrow^+ C'$;*

$$\begin{tikzcd}
c \arrow[r] \arrow[d, "{[\![]\!]}"'] & c' \arrow[d, "{[\![]\!]}"] \\
C \arrow[r, "+"] & C'
\end{tikzcd}$$

3. *or there exists c'', C' such that $[\![c'']\!] = C'$ and $C \longrightarrow^+ C'$;*

$$\begin{tikzcd}
c \arrow[r] \arrow[d, "{[\![]\!]}"'] & c' \arrow[r] & c'' \arrow[d, "{[\![]\!]}"] \\
C \arrow[rr, "+"] & & C'
\end{tikzcd}$$

4. *or $[\![c']\!] = C$ directly, and $\mu(c) > \mu(c')$*

$$\begin{tikzcd}
c \arrow[r, "\mu\searrow"] \arrow[d, "{[\![]\!]}"'] & c' \arrow[dl, "{[\![]\!]}"] \\
C &
\end{tikzcd}$$

Proof By case analysis on the applied rule.

Context. By lemma 41.

IM. By proposition 32, noting that the number of $\mathsf{let\ rec}$ nodes decreases by one when applying the rule.

EM. $c = b_v \vdash \mathsf{let\ rec}\ b\ \mathsf{in}\ e$ and $c' = b_v, b \vdash e$. Let us now define C_1 by $\emptyset \vdash [\![e]\!]$ if b is not evaluated, and $[\![e]\!]^{\mathrm{TOP}}$ otherwise. Then $[\![c]\!]^{\mathrm{TOP}} = \mathit{TOP}(b_v) \circ [\![b]\!]^{\mathrm{TOP}}[C_1]$. Let $b = {b_v}', b'$, where b' does not begin with a value. We have $[\![c]\!]^{\mathrm{TOP}} = \mathit{TOP}(b_v) \circ \mathit{TDum}(b') \circ \mathit{TOP}({b_v}') \circ \mathit{TUp}(b')[C_1]$. But the substitution of the context $\mathit{TDum}(b')$ does not affect $\mathit{TOP}(b_v)$ and conversely the substitution of $\mathit{TOP}(b_v)$ does not affect $\mathit{TDum}(b')$, so the two contexts commute. But then $\mathit{TOP}(b_v)$ is next to $\mathit{TOP}({b_v}')$. Let η be the substitution of $\mathit{TDum}({b_v}')$. It does not affect $\mathit{TOP}(b_v)$, by the side condition to the EM rule, so $\mathit{TOP}(b_v) \circ \mathit{TOP}({b_v}') = \eta \circ \mathit{TOP}(b_v) \circ \mathit{TOP}({b_v}')$, which by corollary 10 is equal to $\mathit{TOP}(b_v, {b_v}')$. Therefore, $[\![c]\!]^{\mathrm{TOP}} = \mathit{TDum}(b') \circ \mathit{TOP}(b_v, {b_v}') \circ \mathit{TUp}(b')[C_1] = [\![b_v, b]\!]^{\mathrm{TOP}}[C_1]$. The number of $\mathsf{let\ rec}$ nodes again decreases by one.

Subst. $c = \mathbb{E}[\mathbb{N}[x]]$, $c' = \mathbb{E}[\mathbb{N}[v]]$, and $\mathbb{E}(x) = v$. Let $\Psi = [\![\mathbb{E}]\!]^{\mathrm{TOP}} = \Theta \vdash \Phi[\sigma]$.

- If v is a variable y, then $[\![v]\!]^{\mathrm{TOP}} = \emptyset \vdash y$, and by proposition 31, $\sigma(x) = y\{\sigma\}$, so $[\![c]\!]^{\mathrm{TOP}} = [\![c']\!]^{\mathrm{TOP}}$. But, the depth of the binder of the hot variable, from the depth of $x = y$ in $\mathbb{E}$, becomes either an upper $y = v'$ definition, or the depth 0, if y is not defined by $\mathbb{E}$, so $\mu(c) > \mu(c')$.
- If c' is faulty, i.e. either $\mathbb{N} = \Box v'$ and v is a record, or $\mathbb{N} = \Box.X$ and v is a function or a record with no X field, then C is faulty too.

- If $v = \lambda y.e$ and $\mathbb{N} = \Box v'$, then $c' \longrightarrow c'' = \mathbb{E}[\mathsf{let\ rec}\ y = v'\ \mathsf{in}\ e]$.
  Let $[\![v']\!]^{\mathrm{TOP}} = \Theta'_v \vdash V'$. Let $\phi = \Theta'_v \vdash \Box[id]$. We have $C = \Psi \circ \phi[lV']$.
  But by proposition 31, the location $l = \sigma(x)$ is such that $\Theta(l) = \lambda y.[\![e]\!]$. Therefore, C reduces by rule Context (Beta) to $\Psi \circ \phi[[\![e]\!]\{y \mapsto V'\}]$. By lemma 39, this reduces to $\Psi \circ \phi[[\![e]\!]^{\mathrm{TOP}}\{y \mapsto V'\}]$.
  Let now $\phi' = \phi \circ \{y \mapsto V'\}$. The obtained configuration can be written $\Psi \circ \phi'[[\![e]\!]^{\mathrm{TOP}}]$.
  But $\mathit{TOP}(y = v') = \Theta'_v \vdash \Box[y \mapsto V'] = \phi'$, so $[\![\mathsf{let\ rec}\ y = v'\ \mathsf{in}\ e]\!]^{\mathrm{TOP}} = \phi'[[\![e]\!]^{\mathrm{TOP}}]$, and the obtained term can also be written $[\![\mathbb{E}]\!]^{\mathrm{TOP}}[[\![\mathsf{let\ rec}\ y = v'\ \mathsf{in}\ e]\!]^{\mathrm{TOP}}]$, which by proposition 29, reduces to $[\![\mathbb{E}[\mathsf{let\ rec}\ y = v'\ \mathsf{in}\ e]]\!]^{\mathrm{TOP}}$, which is exactly $[\![c'']\!]^{\mathrm{TOP}}$.
- If $v = \{s_v\}$, $\mathbb{N} = \Box.X$, with $X \in dom(s_v)$, then $c' \longrightarrow c'' = \mathbb{E}[s_v(X)]$.
  By hypothesis, c is in **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 λ_{alloc}, and $[\![v]\!]^{\mathrm{TOP}} = \{l \mapsto \{s_v\}\} \vdash l$.
  By proposition 31, we have $\sigma(x) = l$ and $\Theta(l) = \{s_v\}$. We have $[\![c]\!]^{\mathrm{TOP}} = \Psi[x.X] = \Psi[l.X]$. As c reduces to c', there exists an index i_0 such that $X = X_{i_0}$. So, $[\![c]\!]^{\mathrm{TOP}}$ reduces in one Project step to $\Psi[x_{i_0}]$, which is $[\![\mathbb{E}]\!]^{\mathrm{TOP}}[[\![x_{i_0}]\!]^{\mathrm{TOP}}]$, so by lemma 39, it reduces to $[\![\mathbb{E}[x_{i_0}]]\!]^{\mathrm{TOP}}$, which is exactly the translation of c''.

□

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 $\emptyset \vdash [\![e]\!] \longrightarrow^* [\![a]\!]^{\mathrm{TOP}}$.*
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 $[\![e]\!]^{\mathrm{TOP}}$, 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 λ_{alloc} simulating the one in $\lambda_\circ$, 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. □

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 $[\![a]\!]^{\mathrm{TOP}} = A$.*
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 $[\![\emptyset \vdash e]\!]^{\mathrm{TOP}}$. 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 $\lambda_\circ$, the rule for substitution SUBST allows this, in conjunction with the internal access rule 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 $\lambda_\circ$. 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 λ-graph reduction and environment based evaluation, and different evaluation strategies. No emphasis is put on data representation either.

Equational theories of the λ-calculus with explicit recursion Ariola et al. [7] study a λ-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 $\lambda_\circ$. The semantics of our source language $\lambda_\circ$ 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 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 λ-abstraction).

let rec **for objects and mixin modules** Boudol's construct [12], or Hirschowitz and Leroy's [45], are different from the one of $\lambda_\circ$ 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 let rec $o = \mathit{generator}(o)$ in e. Such definitions are impossible in $\lambda_\circ$. However, $\lambda_\circ$ 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 $\lambda_\circ$. 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 $\lambda_\circ$, 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

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Func. Progr.*, 1(4):375–416, 1991.

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

[4] Davide Ancona, Sonia Fagorzi, Eugenio Moggi, and Elena Zucca. Mixin modules and computational effects. Technical report, DISI, Università di Genova, 2002.

[5] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In Gopalan Nadathur, editor, *Princ. and Practice of Decl. Prog.*, volume 1702 of *LNCS*, pages 62–79. Springer-Verlag, 1999.

[6] Davide Ancona and Elena Zucca. A calculus of module systems. *J. Func. Progr.*, 12(2):91–132, 2002.

[7] Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):95–178, 2002.

[8] Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inf. and Comp.*, 139:154–233, 1997.

[9] Zine-El-Abidine Benaissa, Pierre Lescanne, and Kristoffer H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Prog. Lang., Impl., Logics, and Programs*, 1996.

[10] Viviana Bono, Lorenzo Bettini, and Betti Venneri. Subtyping mobile classes and mixins. FOOL'03.

[11] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping for extensible, incomplete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

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

[13] Gérard Boudol. The recursive record semantics of objects revisited. Research report 4199, INRIA, 2001. Preliminary version presented at ESOP'01, LNCS 2028.

[14] S. Boulmé. *Spécification d'un environnement de calcul formel certifié*. PhD thesis, Université Paris VI, 2000.

[15] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Modules, objets et calcul formel. In *Journ. Franc. des Lang. Applicatifs*, 1999.

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

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

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

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

[20] Luca Cardelli, J. Donahue, L. Glassman, M. Jordan an B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation Systems Research Center, 1989.

[21] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990. Also available as research report 56, DEC Systems Research Center.

[22] Thierry Coquand and Gérard Huet. The calculus of Constructions. *Inf. and Comp.*, 76(2/3):95–120, 1988.

[23] Judicaël Courant. An applicative module calculus. In *Theory and Practice of Software Development 97*, Lecture Notes in Computer Science, pages 622–636, Lille, France, April 1997. Springer-Verlag.

[24] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[25] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[26] Karl Crary, Robert Harper, Perry Cheng, Leaf Petersen, and Chris Stone. Transparent and opaque interpretations of datatypes. Technical Report CMU–CS–98–177, Carnegie Mellon University, 1998.

[27] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Prog. Lang. Design and Impl.*, pages 50–63. ACM Press, 1999.

[28] Derek R. Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *symp. Principles of Progr. Lang.*, 2003.

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

[30] Dominic Duggan. A mixin-based, semantics-based approach to implementing modular reusable domain-specific programming languages. In *Europ. Conf. on Object-Oriented Progr.*, 2001.

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

[32] Dominic Duggan and Constantinos Sourelis. Recursive modules and mixin-based inheritance. Unpublished draft, 2001.

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

[34] Matthew Flatt. PLT MzScheme: language manual. Technical Report TR97-280, Rice University, 1997.

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

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

[37] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'État, Université Paris VII, 1972.

[38] Neal Glew. A theory of second-order trees. In Daniel Le Métayer, editor, *Europ. Symp. on Progr.*, volume 2305 of *LNCS*, pages 147–161. Springer-Verlag, 2002.

[39] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Trans. Prog. Lang. Syst.*, 22(6):1037–1080, 2000.

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

[41] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th symp. Principles of Progr. Lang.*, pages 341–354. ACM Press, 1990.

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

[43] Robert Harper and Chris Stone. An interpretation of standard ML in type theory. Technical Report CMU–CS–97–147, Carnegie Mellon University, 1997.

[44] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical report, Kyoto University, 1997.

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

[46] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. Long version of [45], 2002.

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

[48] Java. `http://java.sun.com`.

[49] Javabeans. `http://java.sun.com/beans`.

[50] Xavier Leroy. *Typage polymorphe d'un langage algorithmique.* Thèse de doctorat, Université Paris VII, 1992.

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

[52] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[53] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[54] Xavier Leroy. A modular module system. *J. Func. Progr.*, 10(3):269–303, 2000.

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

[56] Mark Lillibridge. *Translucent Sums : a Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[57] M.D. McIlroy. Mass produced software components. Report on a Conference of the NATO Science Committee, 1968.

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

[59] John C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial intelligence and mathematical theory of computation*, pages 305–330. Academic Press, 1991.

[60] Martin Odersky, Vincent Cremet, Christine Röcl, and Matthias Zenger. A nominal theory of objects with dependent types. FOOL'03.

[61] Chris Okasaki. *Purely Functional Data Structures*, chapter 10. Cambridge University Press, 1998.

[62] Virgile Prevosto and Damien Doligez. Algorithms and proofs inheritance in the FOC language. *??*, 2002.

[63] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[64] Kristoffer H. Rose. Explicit cyclic substitution. Unpublished, March 1993.

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

[66] Claudio V. Russo. Recursive structures for Standard ML. In *Int. Conf. on Functional Progr.*, pages 50–61, 2001.

[67] Masahiko Sato and Rod Burstall. Explicit environments. In *First International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*, 1998.

[68] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. In *Int. Conf. on Typed Lambda Calculi and Appl.*, 1999.

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

[70] Peter Sewell. Modules, abstract types, and distributed versioning. In *symp. Principles of Progr. Lang.*, 2001.

[71] Zhong Shao. Transparent modules with fully syntactic signatures. Technical Report YALEU/DCS/TR–1181, Yale University, 1999.

[72] Chris Stone. *Singleton kinds and singleton types.* PhD thesis, Carnegie Mellon University, 2000.

[73] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2 edition, 1991.

[74] Clemens A. Szyperski. Import is not inheritance; why we need both: Modules and classes. In *Europ. Conf. on Object-Oriented Progr.*, volume 615 of *LNCS*, pages 19–32, 1992.

[75] J. B. Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. Long version of [76].

[76] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Europ. Symp. on Progr.*, volume 1782 of *LNCS*, pages 412–428. Springer-Verlag, 2000.

[77] Benjamin Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris VII, 1994.

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