

Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types

Benoît Montagu

▶ **To cite this version:**

Benoît Montagu. Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types. Programming Languages [cs.PL]. Ecole Polytechnique X, 2010. English. NNT : . tel-00550331

HAL Id: tel-00550331
https://pastel.hal.science/tel-00550331

Submitted on 27 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée à
l'École Polytechnique

pour obtenir le titre de
Docteur de l'École Polytechnique
Spécialité
Informatique

Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts

---

soutenue par
Benoît Montagu
le 15 Décembre 2010

Jury	
Président	M. Gilles Dowek
Rapporteurs	M. Jacques Garrigue
	M. Claudio V. Russo
Examinateurs	M^{me} Catherine Dubois
	M. Hugo Herbelin
Directeur	M. Didier Rémy

Projet Gallium — INRIA Paris-Rocquencourt

Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types

Abstract

This thesis explains how the adjunction of three features to System F^ω allows writing programs in a modular way in an explicit system *à la* Church, while keeping a style that is similar to ML modules. The first chapter focuses on open existential types, which provide a way to consider existential types without scope restrictions: they permit to organize programs in a more flexible manner. The second chapter is devoted to the study of singleton kinds, which model type definitions: in this framework, we give a simple characterization of type equivalence, that is based on a confluent and strongly normalizing reduction relation. The last chapter integrates the two previous notions into a core language equipped with a subtyping relation: it greatly improves the modularity of F^ω to a level that is comparable with the flexibility of ML modules. A translation from modules to this core language is defined, and is used to precisely compare the two languages.

Résumé

Cette thèse décrit comment l'ajout de trois ingrédients à Système F^ω permet d'écrire des programmes de façon modulaire dans un système explicite à la Church, tout en gardant un style proche des modules de ML. Le premier chapitre s'intéresse aux types existentiels ouverts, qui confèrent la possibilité d'utiliser des types existentiels sans restriction de portée : cela offre une plus grande flexibilité dans l'organisation des programmes. Le deuxième chapitre est consacré à l'étude des kinds singletons, qui modélisent les définitions de types : dans ce cadre, on donne une caractérisation simple de l'équivalence de types, fondée sur une relation de réduction confluente et fortement normalisante. Le dernier chapitre intègre les deux notions précédentes dans un langage noyau muni d'une relation de sous-typage : cela apporte à F^ω un gain de modularité important, de niveau comparable à celui des modules de ML. Une traduction des modules vers ce langage est esquissée, permettant une comparaison précise des deux langages.

Remerciements

En premier lieu, je souhaite dire un grand merci aux deux rapporteurs de cette thèse, Claudio V. Russo et Jacques Garrigue, qui ont su faire abstraction des typos, initialement très nombreuses, pour lire en profondeur ce document : la qualité et la minutie de leur travail et des suggestions et critiques qu'ils m'ont adressées ont été extrêmement appréciables. Je tiens également à remercier les autres membres du jury, qui m'ont fait l'honneur d'accepter d'examiner ces travaux.

Ensuite, mon directeur de thèse, Didier Rémy, pour son perfectionnisme et sa grande rigueur scientifique, mérite un remerciement particulier. Il m'a aidé à traverser des épreuves difficiles et n'a jamais tari d'encouragements à mon égard. Plus important (et plus difficile ?) il a été capable de me supporter pendant plus de trois ans !

J'ai eu le plaisir d'être accueilli au sein du projet Gallium, qui constitue un cadre de travail proche de l'idéal. J'adresse donc un grand merci aux membres du projet et de son projet frère Moscova. Je dois avouer que les pauses café, souvent d'une grande richesse culturelle, resteront dans mon souvenir pour leur caractère singulier et chaque fois renouvelé.

Un merci spécial aux post-doctorants, doctorants et stagiaires qui m'auront accompagnés ou que j'aurai croisés dans cette aventure : merci donc à (dans un ordre indéterminé) Yann, Boris, Zaynah, Benoît, Jean-Baptiste, Keiko, Nicolas, Arthur, Tahina, Jade, Vivien, Gabriel, Alexandre, Jonathan et Julien, qui ont su, chacun à leur manière, me faire lever la tête pour sortir de mes preuves et autres lettres grecques.

Je ne peux oublier Stephen, qui lui aussi, a eu le mérite de me supporter pendant plus de trois années successives. Je ne sais toujours pas ce qu'il pouvait penser lorsqu'il me voyait passer certains de mes soirées, nuits ou week-ends à taper sur un ordinateur tantôt des lignes d'OCaml, tantôt de Coq, tantôt de LaTeX. L'image que j'ai pu lui donner, bien malgré moi, de la recherche mérite sans aucun doute que l'on s'y attarde.

Cette énumération serait terriblement incomplète si je n'accordais pas un remerciement particulier aux coureurs infatigables de chez RAP : la frénésie sportive de Philippe et la combativité de Mathieu, ainsi que leur bonne humeur, sont extrêmement contagieuses, au point de donner l'envie de courir un marathon, et d'entraîner dans nos courses certains doctorants de chez Gallium ou Secret. L'exercice de la thèse et l'épreuve du marathon, d'ailleurs, partagent de manière intéressante des traits communs. Un grand merci à eux pour cette heureuse contamination.

Je n'oublie pas mes amis de l'X ou de prépa restés anonymes, qui m'ont permis de garder les pieds sur terre en me faisant garder à l'esprit qu'un monde existe en dehors de la thèse.

Aurélie, avec qui j'ai partagé des « week-ends rédaction » mais aussi beaucoup plus, a tenu une place très particulière durant ces dernières années. Reçois toute mon affection.

Enfin, mes parents et mon frère, que je n'ai pas vus assez souvent ces dernières années, m'ont entouré de leurs encouragements et de leurs soins. Merci à vous trois.

Les travaux présentés dans ce document ont été effectués dans le cadre d'une allocation de recherche AMX et du contrat ANR U3CAT.

« Ce n'est qu'en essayant continuellement que l'on finit par réussir. Ou en d'autres termes : plus ça rate, plus on a de chances que ça marche. »

(Principe de base de la logique Shadok*)*

Contents

1 Introduction 1
- 1.1 A short overview of ML modules . . . 1
- 1.2 Russo's interpretation of modules . . . 4
- 1.3 Problematics and outline . . . 5
- 1.4 Published work . . . 5

2 Open existential types 7
- 2.1 Existential types in System F . . . 7
- 2.2 A core calculus with open existential types . . . 8
  - 2.2.1 More atomic constructs for existential types . . . 8
  - 2.2.2 The appearance of recursive types . . . 14
  - 2.2.3 About coercibility . . . 14
- 2.3 A definition for Core $F^{\forall}$ . . . 15
  - 2.3.1 A more restrictive zipping . . . 15
  - 2.3.2 Syntax . . . 16
  - 2.3.3 Typing rules . . . 19
  - 2.3.4 Reduction semantics . . . 19
- 2.4 Soundness . . . 23
  - 2.4.1 Basic syntactic lemmas . . . 23
  - 2.4.2 Main syntactic lemmas . . . 24
  - 2.4.3 Properties of coercibility . . . 25
  - 2.4.4 Properties of results and ϵ-reductions . . . 27
  - 2.4.5 Type soundness . . . 27
  - 2.4.6 A mechanized proof of soundness . . . 28
  - 2.4.7 Type erasure semantics . . . 29
- 2.5 Adequacy with System F . . . 29
  - 2.5.1 From F to $F^{\forall}$ . . . 30
  - 2.5.2 From $F^{\forall}$ to F . . . 30
  - 2.5.3 The logical facet . . . 36
- 2.6 Extensions of $F^{\forall}$ . . . 36
  - 2.6.1 Weakening . . . 36
  - 2.6.2 More liberal equations . . . 37
  - 2.6.3 Double vision . . . 40
  - 2.6.4 Recursive types . . . 41
  - 2.6.5 Recursive values . . . 42
  - 2.6.6 Soundness of the extensions . . . 43
- 2.7 Related work . . . 49
- 2.8 Conclusion and future work . . . 51
  - 2.8.1 Limitations of $F^{\forall}$ . . . 52
  - 2.8.2 Future work . . . 52

3 Type definitions and singleton kinds 55
3.1 Singleton kinds: Harper-Stone system . . . 56
3.1.1 Harper-Stone's system: definitions . . . 56
3.1.2 Examples . . . 58
3.1.3 Harper-Stone's system: properties . . . 60
3.1.4 Harper-Stone normalization algorithm and decidability result . . . 62
3.2 Goals of this chapter . . . 64
3.3 Preliminary results: some composition properties for rewriting systems . . . 65
3.4 Warm-up: the simply-typed case . . . 67
3.4.1 Definitions . . . 68
3.4.2 Subject reduction . . . 70
3.4.3 Confluence and strong normalization . . . 70
3.4.4 Adequacy . . . 72
3.5 Small-step extensional equivalence for singleton types . . . 76
3.5.1 Definition . . . 76
3.5.2 Translation into the simply typed λ-calculus . . . 79
3.5.3 Subject reduction . . . 80
3.5.4 Confluence and strong normalization . . . 81
3.5.5 Properties of expansors . . . 82
3.5.6 Soundness of convertibility . . . 84
3.5.7 Completeness of convertibility . . . 85
3.5.8 Adequacy . . . 98
3.5.9 Insertions of expansors and minimal kinds . . . 99
3.5.10 A second reading of Stone-Harper's normalization algorithm . . . 103
3.6 Related work . . . 104
3.7 Future work . . . 105

4 A tentative design 107
4.1 Definitions . . . 107
4.1.1 Terms . . . 108
4.1.2 Types and kinds . . . 108
4.1.3 Coercibility . . . 112
4.1.4 A powerful notion of subtyping . . . 114
4.1.5 Environments . . . 117
4.1.6 Dynamic semantics . . . 118
4.1.7 Conjectures . . . 118
4.2 Examples and remarks . . . 119
4.2.1 Local definitions . . . 119
4.2.2 Renaming or relocation of existential items . . . 119
4.2.3 Phase-split style . . . 120
4.3 Comparisons . . . 121
4.3.1 $\mathsf{F}^{\forall\omega}_{S\leqslant}$ vs. System F^{ω} . . . 121
4.3.2 $\mathsf{F}^{\forall\omega}_{S\leqslant}$ vs. ML . . . 122
4.4 Conclusion and future work . . . 133

5 Conclusion 135

List of Figures

1.1 Sample code: ordered types and finite sets. . . . 2

2.1 Open existential constructs. . . . 13
2.2 Zipping of bindings (preliminary definition). . . . 13
2.3 Coercibility (Core $F^{\forall}$). . . . 14
2.4 Zipping: definition. . . . 16
2.5 Syntax: types, terms, values, and results. . . . 17
2.6 Universally-weakened environment. . . . 17
2.7 Wellformedness of types. . . . 17
2.8 Wellformedness of environments. . . . 17
2.9 Typing rules of the core system. . . . 18
2.10 Coercible types. . . . 18
2.11 Example of extrusion. . . . 20
2.12 Reduction rules. . . . 22
2.13 Type normalization. . . . 26
2.14 Type erasure. . . . 29
2.15 Encoding from F to $F^{\forall}$. . . . 30
2.16 Encoding from $F^{\forall}$ to F, stage 1: recovering packs. . . . 30
2.17 Encoding from $F^{\forall}$ to F, stage 2: recovering unpack s (congruence rules are omitted). . . . 31
2.17 Encoding from $F^{\forall}$ to F, stage 2 (continued). . . . 32
2.18 Zipping of sets of bindings 38
2.19 Closing mutually recursive type equations. . . . 42
2.20 Typing rules of the extended system. . . . 44
2.21 Weaker environments. . . . 44
2.22 Wellformed environments. . . . 45
2.23 Wellformed types. . . . 45
2.24 Coercible types (co-inductive definition). . . . 45
2.25 Similar types. . . . 46

3.1 Wellformed environments and wellformed kinds. . . . 57
3.2 Subkinding and kind equivalence. . . . 57
3.3 Wellformed types. . . . 58
3.4 Type equivalence. . . . 59
3.5 Natural kind. . . . 62
3.6 Head reduction. . . . 63
3.7 Head normalization. . . . 63
3.8 Type normalization. . . . 63
3.9 Path normalization. . . . 63
3.10 Kind normalization. . . . 63
3.11 $\beta\pi\eta$-equality. . . . 69
3.12 Minimal kinds. . . . 99
3.13 Natural kinds. . . . 102

4.1 Typing rules of $F^{\forall\omega}_{\mathcal{S}\leqslant}$. . . . 109
4.2 Wellformed types. . . . 111
4.3 Wellformed kinds. . . . 112
4.4 Subkinding. . . . 112
4.5 Equivalent kinds. . . . 112
4.6 Type equivalence. . . . 113
4.7 Coercible types. . . . 114
4.8 Coercible kinds. . . . 115
4.9 Subtyping rules. . . . 116
4.10 Wellformed environments. . . . 117
4.11 Environment weakening. . . . 117
4.12 Zipping environments. . . . 118
4.13 Extending System F^{ω} in three directions, and translating back to System F. . . . 121
4.14 Syntax of an idealized ML module language. . . . 123
4.15 Translation of module expressions. . . . 125
4.16 Translation of structure bindings. . . . 127
4.17 Translation of signatures. . . . 128
4.18 Translation of signature bindings. . . . 129
4.19 Translation of first-class modules. . . . 132

Chapter 1

Introduction

Today's hardware and software probably belong to the most complex creations and inventions of mankind. This complexity may result from the possible intrinsic sophistication of the employed algorithms, but also from the elaborateness of the architecture of programs. Rigor, organization and separation of independent components become a necessity, with programs gaining in size: *modular programming is the key to the successful development of large pieces of programs*.

Programming languages can bring programmers a valuable help, and encourage modular development. The module systems from the family of the ML language [MTHM97] are powerful languages that help *programming in the large*. They have been successfully implemented and used for dozens of years in the languages Standard ML [SML, RRS00] and OCaml [LDG+10], to cite only a few.

Other techniques, such as object orientation, mixins, traits, or type classes, also permit modular development and code reuse, but will not be treated in this document.

1.1 A short overview of ML modules

ML modules constitute a layer on top of a base language. For this reason, modules are *second class* citizens: they are not considered as values of the base language, and, conversely, terms from the base languages are not module expressions. In this section, we review the main ingredients on which the ML module system is built. Pierce's book contains a thorough introduction to ML modules [HP05]. An exhaustive and more technical presentation of the works on ML modules can be found in the introduction of Dreyer's thesis [Dre05b].

Structures and signatures Structures are the base elements of modules: they contain the pieces of code; in some narrow sense, they are similar to compilation units in other languages. Interfaces, or signatures, describe the specifications of modules, that is, they specify which values, functions, types and modules are exported. For instance, the `OrderedType` signature of Figure 1.1 on the following page describes every module that has a type component `t` and a function `compare` of type $\texttt{t} \to \texttt{t} \to \texttt{int}$. The module `OrderedInt` implements the interface `OrderedType` where the type `t` is the type `int`. Structures can be nested, so that they permit hierarchical composition of modules. They are very similar to records with structural types and width and depth subtyping.

Type definitions The possibility to define intermediate names for types is given by type components in structures and signatures. They provide not only the possibility to define types within a structure, but also to reuse a definition by exporting it in the interface. This way, type definitions can be shared between modules, but also between the argument of a functor and its body (see the paragraph on functors on page ??). For example, a structure that has the signature `OrderedType` with type $\texttt{t} = \texttt{int}$ exports a type component `t` that is equal to the type `int`.

```
(* Interface of ordered types *)
module type OrderedType = sig
  type t
  val compare : t → t → int
end

(* Integer as an ordered type *)
module OrderedInt : OrderedType with type t = int
= struct
  type t = int
  let compare x y =
    if x < y then -1
    else if x = y then 0
    else 1
end

(* Interface of sets *)
module type Set = sig
  type t   (* the type of sets *)
  type elt   (* the type of their elements *)
  val empty : t
  val is_empty : t → bool
  val add : elt → t → t
  val remove : elt → t → t
  val union : t → t → t
  val fold : ∀α.(elt → α → α) → t → α → α
  ...
end

(* Interface of sets of ordered types *)
module type OrderedSet = sig
  include Set
  val min_elt, max_elt : t → elt
end

(* Interface of sets of integers *)
module type IntSet = OrderedSet with type elt = int

(* Generic implementation of finite sets over ordered types *)
module MakeSet(O : OrderedType) : OrderedSet with type elt = O.t
(* The type of sets is abstract *)
= struct
  (* Implementation of finite sets *)
  ...
end

(* Sets of integers, built from the MakeSet functor *)
module IntSet = MakeSet(OrderedInt)
```

Figure 1.1: Sample code: ordered types and finite sets.

Abstract types Hiding type information is possible through the use of abstract types: a type that has been made abstract, or *opaque*, is ensured to be different from any other type, but itself (up to expansion of type definitions). For instance, in Figure 1.1, in the definition of the module `OrderedInt`, if one had not written the constraint with `type t = int`, the signature ascription `OrderedInt : OrderedType` would have rendered the type `OrderedInt.t` abstract, and thus different from the type `int`. One often says that, in this case, the signature ascription has generated a new type `t`, hence the name of *generativity*. Signatures can contain multiple type components, that can be *concrete* (or *transparent*, or *manifest*) while others are *abstract* (or *opaque*). This is the case of the signature `Set` with `type elt = int`, for example: its type component `elt` is concrete, whereas its type component `t` is abstract. This notion of signatures that mix opaque and manifest type components were originally described by Leroy's manifest types [Ler94, Ler96] and, independently, by Lillibridge's transluscent sums [Lil97, HL94]. More recently, singleton kinds [SH06] were used to model them, and were put in practice in the TILT compiler [PCHS01].

Functors The penultimate definition of Figure 1.1 deals with a generic implementation `MakeSet` of the interface `OrderedSet`, which is parametrized by an implementation of `OrderedType`. This is called a *functor*, that is, a function from modules to modules. Functors can be higher order, and are first-class citizens in the layer of modules. The last line of the figure defines an instance of the `MakeSet` functor on integers, to build an implementation of sets of integers. It is worth noting that the `MakeSet` functor has the following module type: `functor (O : OrderedType) → OrderedSet with type elt = O.t`. This type illustrates the *dependency* that exists between the argument of the functor and the type of its result. Functors are essential to build generic libraries and increase the possibility of code reuse. It is important to notice that, although functor types might look like dependent types, the ML module system enjoys a phase distinction property [HMM90]: the type of the body only depends on the *static part*, *i.e.* on the type components, of the argument.

Generative and applicative functors The `MakeSet` functor gives a result that contains a type component `t` (*i.e.* the type of sets), which is abstract. But will every new application of `MakeSet` yield a different type for the type of sets? It depends on whether the functor is considered generative or applicative. If it is considered as a generative functor, then every new application generates a new abstract type. If it is considered as an applicative functor, then the abstract type will depend on the nature of its argument. For instance, in OCaml, if we define a module `OrderedFloat` of ordered `floats` and then define `module FloatSet = MakeSet(OrderedFloat)`, then `FloatSet.t` and `IntSet.t` are incompatible types. In contrast, if we define a second instance of sets of integers with `module IntSet2 = MakeSet(OrderedInt)`, then `IntSet.t` and `IntSet2.t` *are* compatible: indeed, since the functor is applied twice to identical arguments, there is no reason to make the resulting types distinct. The analysis of identity in OCaml is based on the notion of *paths*. Standard ML implements generative functors only; OCaml has applicative functors by default; and Moscow ML has both of them, which are distinguished by their types. Dreyer's thesis [Dre05b] analyses cases where applicative functors cannot enforce enough invariants on structures, although they stay type-safe, and forbids their problematic uses: it is, for example, the case of functors whose bodies have side effects.

First-class modules The two-layers structure of modules, permits to add the power of the ML module system to any other language [Ler00]. However, in the case of ML, this stratification somehow duplicates concepts: there are functions and functors, structures and records, and a form of polymorphism at both levels. It certainly creates issues for newcomers to learn and understand the language. Unifying the two layers rapidly becomes a need: Moscow ML and, very recently, OCaml offer the possibility to embed a module in a package as a value, and to recover a module from a packaged module. This way, one gets *first-class modules*, which greatly extends the expressivity of

the whole language [FG10]. First-class modules make the frontier between the base and the module layers blurry, but the two layers still remain distinct.

1.2 Russo's interpretation of modules

At first sight, modules of ML incorporate notions that are not present in the ML base programming language: generativity, applicativity, opaque type components, dependent types, paths… It seems the distance between the two layers is not narrow.

However, as shown by Russo [Rus99, Rus03], modules can be given non-dependent types: more precisely, types of System F are enough. One can indeed interpret modules as programs typed in System F^ω. The higher order part of the type language is only necessary to encode parametrized types. This interpretation has recently been formalized in Coq [RRD10]. Russo's original interpretation had the semantic objects of the definition of Standard ML [MTHM97] as a target.

The interpretation shares an idea due to Mitchell and Plotkin [MP88], that abstract types are existential types. This has been exploited in other translations of modules into F^ω [Sha98, cS06]. Consequently, abstract type components in a covariant position are interpreted as existential types; when they are in a contravariant position, they are interpreted with universal types. This way, a general schema for the type of functors is

$$\forall\overline{\alpha}.(\tau[\overline{\alpha}] \to \exists\overline{\beta}.\tau'[\overline{\alpha}][\overline{\beta}])$$

where the αs correspond to the abstract type components of the signature of the argument, whereas the βs represent the abstract type components of the signature of the body. Notice that, first, type definitions have been unfolded, and second, that the dependency between the argument and the body is handled externally via the universal quantification. The type of the `SetMake` functor from Figure 1.1 would be translated to some type close to the following one:

$$\forall\alpha_{\texttt{elt}}.\{\texttt{compare} : \alpha_{\texttt{elt}} \to \alpha_{\texttt{elt}} \to \texttt{int}\} \to \exists\beta_{\texttt{t}}.\left\{\begin{array}{l}\texttt{empty} : \beta_{\texttt{t}}\\ \texttt{is_empty} : \beta_{\texttt{t}} \to \texttt{bool}\\ \texttt{add} : \alpha_{\texttt{elt}} \to \beta_{\texttt{t}} \to \beta_{\texttt{t}}\\ \ldots\end{array}\right\}$$

Russo's approach generalizes well to applicative functors: to express the dependency between the abstract types of the result with respect to the types of the arguments, he uses higher-order types. In the case of the `SetMake` functor, its applicative interpretation would have type:

$$\exists(\beta_{\texttt{t}} :: \star \Rightarrow \star)\ \forall\alpha_{\texttt{elt}}.\{\texttt{compare} : \alpha_{\texttt{elt}} \to \alpha_{\texttt{elt}} \to \texttt{int}\} \to \left\{\begin{array}{l}\texttt{empty} : \beta_{\texttt{t}}\ (\alpha_{\texttt{elt}})\\ \texttt{is_empty} : \beta_{\texttt{t}}\ (\alpha_{\texttt{elt}}) \to \texttt{bool}\\ \texttt{add} : \alpha_{\texttt{elt}} \to \beta_{\texttt{t}}\ (\alpha_{\texttt{elt}}) \to \beta_{\texttt{t}}\ (\alpha_{\texttt{elt}})\\ \ldots\end{array}\right\}$$

Russo's approach also extends to first-class modules and to recursive modules [Rus01], and is at the origin of the implementation of modules in Moscow ML.

1.3 Problematics and outline

The formal interpretation into System F^ω [RRD10] showed, on the one hand, that ML modules can be understood as a stylized way of constructing programs in System F^ω, and, in the other hand, that directly constructing them by hand would not be easy. In this thesis, we try to understand why: what, precisely, does System F^ω lack to permit the construction of programs in a way that would be as elegant, comfortable and modular as it would be with the use of modules? Or, to express it differently, what can we add to System F^ω to bring it closer to ML modules?

The organization of this document follows the way our answer to the above question is articulated: two main features are present in ML but not in F^ω. The possibility to handle programs with abstract types (that is, with existential types) in a modular and convenient manner is one of the major differences: this is treated in Chapter 2 by studying $\mathsf{F}^\curlyvee$ (F-zip), a language featuring *open* existential types, that extends System F to allow the unpacking of existentials in an open scope. A large subset of $\mathsf{F}^\curlyvee$ was formalized in the Coq proof assistant. Our second central point of interest is the support for type definitions: in Chapter 3, we focus on this topic and present a method to decide type equivalence in the singleton kind system of Stone and Harper [SH06], that is based on a well behaved reduction relation. Finally, we merge the two systems in Chapter 4, where we also add subtyping: the resulting language $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ (F-zip-full) is then compared with ML through the definition and analysis of an interpretation of ML into $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$.

1.4 Published work

The results from Chapter 2 were published or presented in the following documents.

[1] Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, January 2009

[2] Benoît Montagu and Didier Rémy. Types abstraits et types existentiels ouverts. In Éric Cariou, Laurence Duchien, and Yves Ledru, editors, *Actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, pages 147–148, Université de Pau, Mars 2010

[3] Benoît Montagu. Experience report: Mechanizing core F-zip using the locally nameless approach. Presented at the 5th ACM SIGPLAN Workshop on Mechanizing Metatheory, Baltimore, September 2010

« On nous cache tout, on nous dit rien,
Plus on apprend, plus on ne sait rien,
On nous informe vraiment sur rien. »

(Jacques Dutronc)

Chapter 2

Open existential types

This chapter deals with type abstraction in module-like languages. The possibility to create abstract types is one of the strengths of the ML module system. Indeed, abstract types provide a way to strongly isolate components of a program: a library that exports an abstract type is guaranteed not to leak more information than the one given in its signature, that is, provided by the public functions that manipulate the abstract type. This feature provides security to the implementer of a library: not only can he hide the implementation details, but also protect internal invariants from potential misuses. He is also assured that changing an implementation for an extensionally equivalent one will not break the compilation or the correctness of existing programs that use the library.

For instance, the library of finite sets from the Objective Caml [LDG+10] distribution implements sets as pseudo-balanced binary search trees, but this is hidden thanks to type abstraction. If the implementation was concrete, the property of being a search tree could be broken by a user, and consequently lead to incorrect results. If the implementation was not sealed with the use of abstract types, the invariant of being pseudo-balanced could also be broken, which would degrade performance, whithout affecting correctness.

The type-theoretic interpretation of abstract types is *existential types* [MP88]. They can already be used in System F, but, as we will see, not in an as convenient way as abstract types are used in ML (Section 2.1). This chapter tries to answer the question: *what does System* F *lack to permit modular uses of existential types?* For this purpose, we will introduce in Section 2.2 the language $\mathsf{F}^{\curlyvee}$ (read F-zip), that features *open existential types*: they get rid of some limitations of System F and bring it closer to the practice of ML modules. We prove $\mathsf{F}^{\curlyvee}$'s correctness in Section 2.4 and study its correlation with System F in Section 2.5. A proof of soundness of a large subset of $\mathsf{F}^{\curlyvee}$ was mechanized in the Coq proof assistant. Section 2.6 considers extensions of $\mathsf{F}^{\curlyvee}$ to cope with more advanced features, some of them being related to recursive modules in ML. The chapter ends with a study of related work (Section 2.7 on page 49), among which lies RTG [Dre07a], which shares some similarities in terms of design. The chapter finally concludes with some critical observations (Section 2.8 on page 51).

2.1 Existential types in System F

Mitchell and Plotkin [MP88] showed that abstract types could be understood as existential types. However, it has also been noticed that existential types do not *accurately* model type abstraction in modules, because they lack some modular properties.

In System F, existential types are introduced by the pack construct: provided the term M has some type $\tau'[\alpha \leftarrow \tau]$, the expression $\mathsf{pack}\ \langle \tau, M \rangle\ \mathsf{as}\ \exists\alpha.\tau'$ hides the type information τ, called the *witness* of the existential, from the type of M so that the resulting type is $\exists\alpha.\tau'$.

$$\textsc{Pack}\ \frac{\Gamma \vdash M : \tau'[\alpha \leftarrow \tau]}{\Gamma \vdash \mathsf{pack}\ \langle \tau, M \rangle\ \mathsf{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}$$

Existential types are eliminated by the unpack construct: provided M has type $\exists\alpha.\tau$, the expression $\mathsf{unpack}\ M\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ M'$ binds the type variable α to the witness of the existential and the value

variable x to the *unpacked* term M in the body of M'. The resulting type is the one of M', in which α must not appear free. The reason for this restriction is that otherwise α, which is bound in M', would escape its scope.

$$\frac{\Gamma \vdash M : \exists\alpha.\tau \qquad \Gamma, \alpha, x : \tau \vdash M' : \tau' \qquad \alpha \notin \mathrm{ftv}(\tau')}{\Gamma \vdash \mathsf{unpack}\ M\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ M' : \tau'}\ \textsc{Unpack}$$

From now on, we assume that System F is equipped with records and with the above primitive constructs, although they could also be provided as a well-known syntactic sugar [Rey83] by inversion of control:

$$\begin{aligned}
\exists\alpha.\tau &\triangleq \forall\beta.(\forall\alpha.\tau \to \beta) \to \beta \\
\mathsf{pack}\ \langle \tau, M \rangle\ \mathsf{as}\ \exists\alpha.\tau' &\triangleq (\Lambda\alpha.\,\lambda(x : \tau')\,\Lambda\beta.\,\lambda(k : \forall\alpha.\tau' \to \beta)\,k\ \alpha\,x)\,\tau\,M \\
\mathsf{unpack}\ M\ \mathsf{as}\ \langle \alpha, x : \tau \rangle\ \mathsf{in}\ M' : \tau' &\triangleq (\lambda(x : \exists\alpha.\tau)\,\Lambda\beta.\,\lambda(k : \forall\alpha.\tau \to \beta)\,x\,\beta\ k)\ M\,\tau'\ (\Lambda\alpha.\,\lambda(x : \tau)\,M')
\end{aligned}$$

As the above lines show, the constructs pack and unpack can be defined as combinators.

2.2 A core calculus with open existential types

2.2.1 More atomic constructs for existential types

In this section we split off the constructs for existential types. Indeed, both pack and unpack have modularity problems, but in different ways.

The crucial issue with unpack is *non-locality*: it imposes the same scope to the type variable α and the value variable x, which is emphasized by the non-escaping condition on α. As a result, all uses of the unpacked term must be anticipated. In other words, the only way to make the variable α available in the whole program is to put unpack early enough in the program, which is a non local, hence non modular, program transformation. The reason is that unpack is doing too many things at the same time: opening the existential type, binding the opened value to a variable, and restricting the scope of the fresh type variable.

The problem with pack is mostly *verbosity*: it requires to completely specify the resulting type, thus duplicating type information in the parts that have not been abstracted away. This can be annoying when hiding only a small part of a term, when this term has a very large type. This duplication happens, for instance, when hiding the type of a single field of a large record, or maybe worse, when hiding some type information deeply inside a record. It is caused by the lack of separation between the introduction of an existential quantifier, and the description of which parts of the type must be abstracted away under that abstract name.

In both cases, the lack of *modularity* is related to the lack of *atomicity* of the constructs. Therefore, we propose to split both of them into more atomic pieces, recovering modularity while preserving expressiveness of existential types. To achieve this decomposition, we first need to enrich typing environments with new items.

Richer contexts for typing judgments

The contexts of typing judgments in System F are sequences of items, where an item is either a binding $x : \tau$ from a value variable to a type, which is introduced while typing functions, or a universal type variable $\forall\alpha$, which is introduced while typing polymorphic expressions.

We augment typing environments with two new items: existential type variables $\exists\alpha$ to keep track of the scope of (open) abstract types, and type definitions $\forall(\alpha = \tau)$ to concisely mediate between

the abstract and concrete views of types. That is, typing environments are as follows:

$$\begin{array}{lcl} \Gamma & ::= & \varepsilon \mid \Gamma, b \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Environments)} \\ b & ::= & x : \tau \mid \forall\alpha \mid \forall(\alpha = \tau) \mid \exists\alpha \qquad\qquad\qquad\qquad \text{(Bindings)} \end{array}$$

Wellformedness of typing environments will ensure that no variable is ever bound twice. We shall see below that existential variables have to be treated linearly. It is sensible to consider them as Skolem's constants and to understand type definition bindings as explicit type substitutions. For the moment, we consider environments as sequences modulo reordering of independent items. Their structure will be enriched again in Section 2.6.4 on page 41.

We define the domain of a binding as follows:

$$\mathrm{dom}(x : \tau) \triangleq x \quad \text{and} \quad \mathrm{dom}(\forall\alpha) \triangleq \mathrm{dom}(\forall(\alpha = \tau)) \triangleq \mathrm{dom}(\exists\alpha) \triangleq \alpha$$

The domain of an environment is, as usual, the union of the domains of the bindings it contains. In addition, we may use the following notations for specific domains:

$$\mathrm{dom}^{\simeq}\,\Gamma \triangleq \{\alpha \mid \forall(\alpha = \tau) \in \Gamma\}$$
$$\mathrm{dom}^{\forall}\,\Gamma \triangleq \{\alpha \mid \forall\alpha \in \Gamma\} \quad \mathrm{dom}^{\exists}\,\Gamma \triangleq \{\alpha \mid \exists\alpha \in \Gamma\}$$

We also use in the rest of the chapter the notion of *pure environments*.

Definition 2.2.1 (Purity)**.** When $\mathrm{dom}^{\exists}\,\Gamma$ is empty, we say that Γ is pure and write $\Gamma\,\mathrm{pure}$.

Splitting unpack

We replace unpack with two orthogonal constructs, *opening* and *restriction*, that implement *scopeless unpacking* of existential values and *scope restriction* of abstract types, respectively.

The *opening* $\mathsf{open}\,\langle\alpha\rangle\,M$ expects M to have an existential type $\exists\alpha.\tau$ and *opens* it under the name α, which is *tracked* in the typing environment by the existential item $\exists\alpha$. The rule can also be read bottom-up, treating the item $\exists\alpha$ as a *linear* resource that is consumed by the opening.

$$\textsc{Open}\ \frac{\Gamma \vdash M : \exists\alpha.\tau}{\Gamma, \exists\alpha \vdash \mathsf{open}\,\langle\alpha\rangle\,M : \tau}$$

The fact that, when it is read bottom-up, rule Open makes the environment decrease might seem unusual. Indeed, it imposes that the subterm should not mention the type variable with which it is opened. It follows a subtle control of scope that is already present in works on resourceful λ-calculi: Kesner and Lengrand [KL07] introduce, for instance, an explicit weakening construct, that makes the environment decrease and hence finely controls the scope of a variable.

Interestingly, rule Open also looks dual to the usual rule of type generalization:

$$\textsc{Gen}\ \frac{\Gamma, \forall\alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau}$$

In rule Gen, the quantifier moves downwards from the environment to the type, whereas it happens in the opposite way in rule Open.

The *restriction* $\nu\alpha.\,M$ implements the non-escaping condition of rule Unpack. First, it requires α not to appear free in the type of M, thus enforcing a limited scope. Second, it provides an existential resource $\exists\alpha$ in the environment, that ought to be consumed by some $\mathsf{open}\,\langle\alpha\rangle\,M'$ expression occurring

within M.

$$\textsc{Nu}\ \frac{\Gamma, \exists\alpha \vdash M : \tau \qquad \alpha \notin \mathrm{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\, M : \tau}$$

As with RTG [Dre07a], one may recover unpack as syntactic sugar:

$$\mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M' \triangleq \nu\alpha.\, (\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ M\ \mathsf{in}\ M')$$

This makes explicit the simultaneous operations performed by unpack, which turns out not to be atomic at all: first, it defines a scope for the name α of the witness of the existential type of M; then, it opens M under the name α; finally, it binds the resulting value to x in the remaining expression M'.

The main flaw of unpack, *i.e.* the scope restriction for the abstract name, is essentially captured by the *restriction* construct. However, since the scope restriction has been separated from the unpack, it is not mandatory anymore to put a scope restriction when one wants to open an existential. The abstract type α may now be introduced at the outermost level or given by the typing context and freely made available to the whole program.

Splitting pack

We replace pack with three orthogonal constructs: *existential introduction*, which creates an existential type, *open witness definition*, which introduces a type witness and gives it a name, and *coercion*, which determines which parts of types are to be hidden. We present this separation in two stages: first, we separate the (closed) definition of a witness from the information of which parts are abstracted away; then, we split the definition of a witness again into two pieces that introduce an existential quantifier and the witness, separately.

The *closed witness definition* $\exists(\alpha = \tau)\, M$ introduces an existential type variable α with witness τ (more precisely, the definition $\forall(\alpha = \tau)$) in the environment while typing M, and binds α existentially in the resulting type.

$$\frac{\Gamma, \forall(\alpha = \tau) \vdash M : \tau'}{\Gamma \vdash \exists(\alpha = \tau)\, M : \exists\alpha.\tau'}$$

The *coercion* $(M : \tau)$ replaces the type of M with some *coercible* type τ. The coercibility relation under context Γ, written $\sim$, is the smallest congruence that contains all type-definitions occurring in Γ. A coercion is typically employed to specify where some abstract types should be used instead of their witnesses in the typing of M.

$$\textsc{Coerce}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

The expressiveness of pack is retained, since it can be provided as the following syntactic sugar:

$$\mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as}\ \exists\alpha.\tau' \triangleq \exists(\alpha = \tau)\, (M : \tau') \quad \text{if } \alpha \notin \mathrm{ftv}(M)$$

However, the description of what is being hidden can now be separated from the action of hiding, which avoids repeating some type information. Hence, it makes the creation of existential values, shorter, thus easier, and more maintainable. Indeed, it allows for putting the information of hiding parts of a type deeply inside a term, like in the following record, in which some leaves have been

abstracted away.

$$\begin{array}{l}\exists(\alpha = \mathit{int})\\ \quad \mathsf{let}\ x = \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = 2\}\ \mathsf{in}\\ \quad \mathsf{let}\ y = \{\ell_1 = x\ ;\ \ell_2 = x\}\ \mathsf{in}\\ \quad \{\ell_1 = y\ ;\ \ell_2 = y\}\end{array}$$

The corresponding System F term requires to repeat the type of the whole term.

$$\begin{array}{l}\mathsf{let}\ z =\\ \quad \mathsf{let}\ x = \{\ell_1 = 1\ ;\ \ell_2 = 2\}\ \mathsf{in}\\ \quad \mathsf{let}\ y = \{\ell_1 = x\ ;\ \ell_2 = x\}\ \mathsf{in}\\ \quad \{\ell_1 = y\ ;\ \ell_2 = y\}\ \mathsf{in}\\ \mathsf{pack}\ \langle \mathit{int}, z\rangle\ \mathsf{as}\\ \quad \exists\alpha.\{\ell_1 : \{\ell_1 : \{\ell_1 : \alpha\,;\ \ell_2 : \mathit{int}\}\,;\ \ell_2 : \{\ell_1 : \alpha\,;\ \ell_2 : \mathit{int}\}\}\,;\\ \qquad \ell_2 : \{\ell_1 : \{\ell_1 : \alpha\,;\ \ell_2 : \mathit{int}\}\,;\ \ell_2 : \{\ell_1 : \alpha\,;\ \ell_2 : \mathit{int}\}\}\}\end{array}$$

Moreover, whereas the information of hiding was located at a single place in the $\mathsf{F}^{\forall}$ term, it is duplicated in the F term, as if each leaf of the record had been abstracted independently.

To complete the separation, we now split $\exists(\alpha = \tau)\, M$ further. The *existential introduction* $\exists\alpha.\, M$ introduces an existential type variable in the environment while typing M, and makes α existentially bound in the resulting type. This is the exact counterpart of the open construct.

$$\textsc{Exists}\ \frac{\Gamma, \exists\alpha \vdash M : \tau}{\Gamma \vdash \exists\alpha.\, M : \exists\alpha.\tau}$$

As the rule Nu, it introduces an existential binding in the context. The difference is that the rule Nu does not change the result type, whereas Exists introduces an existential quantifier.

The *open witness definition* $\Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M$ introduces the witness τ for the type variable α: similarly to what is done for $\exists(\alpha = \tau)\, M$, the equation $\forall(\alpha = \tau)$ is added to the context while typing M. In addition, an external name β is provided, in the same way as for the open construct. The internal name α and its equation are only reachable internally, but the witness is denoted externally by the abstract type variable β. The resulting type does not mention the internal name, since it has been substituted for the external one. In other words, the witness definition defines *a frontier between a concrete internal world and an abstract external one*. To keep the system sound, we ensure that a unique witness is hidden behind an external name, hence the use of an existential resource. The typing rule will be refined later (Section 2.6.3 on page 40) to handle the double vision problem [Dre07a].

$$\textsc{Sigma}\ \frac{\Gamma, \forall(\alpha = \tau) \vdash M : \tau'}{\Gamma, \exists\beta \vdash \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M : \tau'[\alpha \leftarrow \beta]}$$

Again, the split construct $\exists(\alpha = \tau)\, M$ can be recovered by the following syntactic sugar:

$$\exists(\alpha = \tau)\, M \ \triangleq\ \exists\beta.\, \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M \quad \text{if } \beta \notin \mathrm{ftv}(\tau, M)$$

It is worth noting that the *open witness definition* corresponds to type abstraction as it is currently done in module languages: a type definition is kept hidden for the outer environment and a type name is generated so that we can refer to it without knowing its concrete definition. Usual existential types are recovered by closing the open witness definition, *i.e.* by hiding the external name for the witness.

As an example, the following piece of program, written in an ML-like syntax, defines an abstract

module of integers:

$$\begin{array}{l}\mathsf{module}\ \mathsf{X} : \mathsf{sig}\ \ \mathsf{type}\ t \quad \mathsf{val}\ z : t \quad \mathsf{val}\ s : t \to t \quad \mathsf{end} = \\ \quad \mathsf{struct}\ \mathsf{type}\ t = \mathit{int} \quad \mathsf{val}\ z = 0 \quad \mathsf{val}\ s = \lambda(\mathrm{x} : \mathit{int})\, \mathrm{x} + 1\ \mathsf{end}\end{array}$$

It provides the zero constant z and the successor function s. The type $\mathrm{X}.t$ is abstract and available in the whole program. Its counterpart in $\mathsf{F}^{\curlyvee}$ is defined hereafter, for which we assume the context $\exists\beta$:

$$\begin{array}{l}\Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int}) \\ \quad (\{z = 0\ ;\ s = \lambda(\mathrm{x} : \mathit{int})\, \mathrm{x} + 1\} : \{z : \alpha\ ;\ s : \alpha \to \alpha\})\end{array}$$

The two pieces of code look similar, except for the fact that the signature ascription has been replaced with an open witness definition. The counterpart of the signature is the type in the coercion. Note that no type component, hence no name for the module, is needed: the counterpart of $\mathrm{X}.t$ is the abstract type β, which is present in the typing context. It is available in the whole program and does not refer to a value variable.

Notice that it is also possible to rewrite this program in two parts, by first creating an existential term and then opening it under the name β. Again, we assume the context $\exists\beta$.

$$\begin{array}{l}\mathsf{let}\ \mathrm{x} = \\ \quad \exists(\alpha = \mathit{int}) \\ \quad\quad (\{z = 0\ ;\ s = \lambda(\mathrm{x} : \mathit{int})\, \mathrm{x} + 1\} : \{z : \alpha\ ;\ s : \alpha \to \alpha\})\ \mathsf{in} \\ \mathsf{open}\ \langle\beta\rangle\ \mathrm{x}\end{array}$$

It has essentially the same effect: in fact the latter will reduce to the former. It shows however that the mechanisms for type abstraction and opening of existentials are the same.

Generative functors

Following Russo [Rus03], generative functors are functions that have a type of the form $\forall\alpha.(\tau_1 \to \exists\beta.\tau_2)$. In ML, generativity is *implicitly* released when the functor is applied. In $\mathsf{F}^{\curlyvee}$, however, the result of the function must be *explicitly opened*, because generativity and evaluation are two separate notions. To get the same result with another fresh type, it suffices to open it again under another name.

A summary of the constructs for open existential types

The different constructs introduced for open existential types are gathered on the diagram of Figure 2.1 on the facing page, which describes their impact on both the typing environment and the resulting type. To increase readability, terms are not printed on the judgments.

The topmost judgment corresponds to a concrete program (of type $\tau[\alpha \leftarrow \tau']$) with an equation $\forall(\alpha = \tau')$ in its environment. With the use of coercions one can mediate to a type τ where the equation has been folded and then go back to the concrete version. Then, using a Σ, we can make the witness abstract by removing the definition from the typing environment and using the external name β instead. In this process, the variable β is marked as existential and the internal name is replaced with the external one. If the external name does not occur free in the resulting type, we can remove the existential item from the environment, without changing the type, to get the bottom right judgment. If this is not the case, we can close the type by transferring the existential quantifier to the type (bottom left judgment). We can then go back by re-opening the existential.

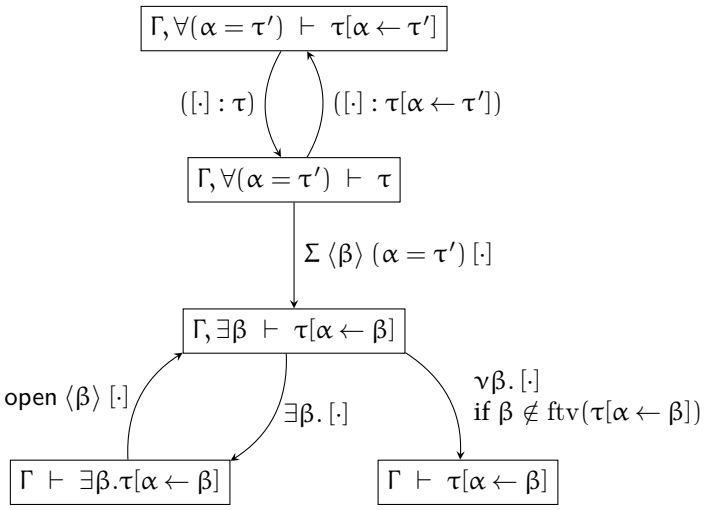


Figure 2.1: Open existential constructs.

$$b \curlyvee b = b \text{ if } b \neq \exists\alpha \quad \exists\alpha \curlyvee \forall\alpha = \exists\alpha \quad \forall\alpha \curlyvee \exists\alpha = \exists\alpha$$

Figure 2.2: Zipping of bindings (preliminary definition).

Linearity to control openings and open witness definitions

As openings and open witness definitions use abstract names given by the environment, one must be careful to avoid "abstraction capture", as in the following (ill-typed) example.

$$\begin{array}{l}\mathsf{let}\ f = \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\ (\lambda(z : \mathit{int})\, z + 1 : \alpha \to \alpha)\ \mathsf{in}\\ \mathsf{let}\ x = \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{bool})\ (\mathit{true} : \alpha)\ \mathsf{in}\ f\ x\end{array}$$

Here, f and x result from two different openings under the same name β. Hence, f and x are assigned types $\beta \to \beta$ and β, respectively, using the *same* abstract name β. However, each branch uses a different witness for β (*int* and *bool* respectively). This yields to the unsound application $f\ x$, which evaluates to $\mathit{true} + 1$.

To prevent abstraction capture, it suffices that *every name β be used in exactly one opening or open witness definition under name β*. This may be achieved by treating the existential items of the typing environment in a *linear* way. As usual in the literature, linearity can easily be enforced in typing rules by a *zipping* operation that describes how typing environments of the premises must be combined to form the one of the conclusion. We give in Figure 2.2 and in this paragraph a preliminary definition of zipping to convey the intuition. It will be completed later. Zipping is a binary operation $(\cdot \curlyvee \cdot)$ that proceeds by zipping individual bindings pointwise. For all items but existential type variables, zipping requires the two facing items to be identical, as usual. The interesting case is when one of the two items is an existential variable $\exists\alpha$: the intuition is that, in this case, the other item must be the universal variable $\forall\alpha$, hence the *zipper* image. This ensures that an existential variable in the conclusion can only be used up in one of the premises. Zipping can also be explained in terms of internal and external choice: the side that makes use of $\exists\alpha$ will make an internal choice by giving the witness. Therefore the other side *must* consider the choice of the witness as external, which is why it is given the item $\forall\alpha$.

Note that an equivalent presentation, using two contexts, one of them being linear, is also feasible.

$$\textsc{Coerce-Eq}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \forall(\alpha = \tau) \in \Gamma}{\Gamma \vdash \alpha \sim \tau} \qquad \textsc{Coerce-Refl}\ \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \tau \sim \tau} \qquad \textsc{Coerce-Sym}\ \frac{\Gamma \vdash \tau_2 \sim \tau_1}{\Gamma \vdash \tau_1 \sim \tau_2} \qquad \textsc{Coerce-Trans}\ \frac{\Gamma \vdash \tau_1 \sim \tau_2 \qquad \Gamma \vdash \tau_2 \sim \tau_3}{\Gamma \vdash \tau_1 \sim \tau_3}$$

Rules for congruence are omitted.

Figure 2.3: Coercibility (Core $\mathsf{F}^{\curlyvee}$).

However, the current presentation makes extensions easier to define.

2.2.2 The appearance of recursive types

The above idea of zipping is unfortunately too generous: it introduces recursive types through the back door. Indeed the decomposition of unpack into opening and restriction opens up the way to recursive types, because it allows one to reference an abstract type variable before its witness has been given. Recursive types can appear through type abstraction, *i.e.* through openings or open witness definitions, in two ways.

We call *internal recursion* the first way, which is highlighted by the following example:

$$\exists\beta.\, \mathsf{let}\ x = \exists(\alpha = \beta \to \beta)\, M\ \mathsf{in}\ \mathsf{open}\ \langle\beta\rangle\ x$$

The abstract type variable β is used in a witness to define x which is then opened under the name β. By reducing this expression we get $\exists\beta.\, \mathsf{open}\ \langle\beta\rangle\ \exists(\alpha = \beta \to \beta)\, M$, which leads us to the recursive equation $\beta = \beta \to \beta$.

We call *external recursion* the second way, which is hereafter exemplified:

$$\begin{aligned}\exists\beta_1.\, \exists\beta_2.\, \{\ \ell_1 &= \Sigma\ \langle\beta_1\rangle\ (\alpha_1 = \beta_2 \to \beta_2)\, M_1\ ; \\ \ell_2 &= \Sigma\ \langle\beta_2\rangle\ (\alpha_2 = \beta_1 \to \beta_1)\, M_2\ \}\end{aligned}$$

The above piece of code is a pair whose components have been abstracted away and the witnesses are mutually dependent. If we remove the type abstractions we get the recursive equation system $\beta_1 = \beta_2 \to \beta_2$ and $\beta_2 = \beta_1 \to \beta_1$.

Notice that recursive types never arise when using System F's unpack. Consider the following piece of code, where C_1 and C_2 denote contexts:

$$\nu\alpha.\, C_2[\mathsf{let}\ x = C_1[\mathsf{open}\ \langle\alpha\rangle\ M_1]\ \mathsf{in}\ M_2]$$

If we consider this program as an unpack, then the contexts C_1 and C_2 are empty. Consequently, α cannot occur free in C_1 or C_2. By splitting unpack, however, this restriction has been waived.

2.2.3 About coercibility

Coercibility $\Gamma \vdash \cdot \sim \cdot$ is nothing more than a congruence, that is parametrized by a set of axioms, *i.e.* by the equations $\forall(\alpha = \tau)$ present in the context Γ.

To illustrate this fact, we can encode coercibility judgments into functions at the term-level, that use primitive constructs for folding and unfolding equations: assume we have two constructs $\mathsf{fold}^{\alpha}\ M$ and $\mathsf{unfold}^{\alpha}\ M$ with the corresponding typing rules

$$\frac{\Gamma \vdash M : \tau \qquad \forall(\alpha = \tau) \in \Gamma}{\Gamma \vdash \mathsf{fold}^{\alpha}\ M : \alpha} \qquad \frac{\Gamma \vdash M : \alpha \qquad \forall(\alpha = \tau) \in \Gamma}{\Gamma \vdash \mathsf{unfold}^{\alpha}\ M : \tau}$$

and the reduction rules

$$\begin{array}{lcl} \mathsf{fold}^{\alpha}\ (\mathsf{unfold}^{\alpha}\ M) & \rightsquigarrow & M \\ \mathsf{unfold}^{\alpha}\ (\mathsf{fold}^{\alpha}\ M) & \rightsquigarrow & M \end{array}$$

Then, one can reify a proof of $\Gamma \vdash \tau_1 \sim \tau_2$ into a pair of functions that are $\beta\eta$-equivalent to the identity, and of respective type $\tau_1 \to \tau_2$ and $\tau_2 \to \tau_1$.

For instance, Coerce-Refl is encoded as the pair $(\lambda(x:\tau)\,x, \lambda(x:\tau)\,x)$, while Coerce-Sym swaps the two encodings, and Coerce-Trans composes them. The rule Coerce-Eq is translated as the pair $(\lambda(x:\alpha)\,\mathsf{unfold}^{\alpha}\ x, \lambda(x:\tau)\,\mathsf{fold}^{\alpha}\ x)$. Then, the congruence rules are constructed by extensionality. We provide the encodings as the rules Coerce-Arrow and Coerce-Exists as examples.

Assume that (c_1, c_1') (*resp.* (c_2, c_2')) is the translation of $\Gamma \vdash \tau_1 \sim \tau_1'$ (*resp.* of $\Gamma \vdash \tau_2 \sim \tau_2'$). Then the encoding of $\Gamma \vdash \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'$ is the pair $(\lambda(f:\tau_1 \to \tau_2)\,\lambda(x:\tau_1')\,c_2\ (f\ (c_1'\ x)), \lambda(f:\tau_1' \to \tau_2')\,\lambda(x:\tau_1)\,c_2'\ (f\ (c_1\ x)))$.

Now assume that (c, c') is the translation of $\Gamma, \forall\alpha \vdash \tau \sim \tau'$. Then the encoding of $\Gamma \vdash \exists\alpha.\tau \sim \exists\alpha.\tau'$ is the pair $(\lambda(x:\exists\alpha.\tau)\,\exists\alpha.\,c\ (\mathsf{open}\ \langle\alpha\rangle\ x), \lambda(x:\exists\alpha.\tau')\,\exists\alpha.\,c'\ (\mathsf{open}\ \langle\alpha\rangle\ x))$.

The other rules of coercibility can be translated as easily. This translation confirms that coercibility is not a difficult notion at all, and that it is a degenerate form of subtyping.

2.3 A definition for Core $\mathsf{F}^{\curlyvee}$

We now present the core of our system, which prevents the appearance of recursive types in a simple manner. We present its semantics and show that its expressive power corresponds exactly to the one of System F. The translation used for this purpose brings interesting insight on the gain of modularity that $\mathsf{F}^{\curlyvee}$ achieves.

2.3.1 A more restrictive zipping

The zipping we defined in Figure 2.2 on page 13 is too liberal in the sense that the introduction of abstract types does not follow the scope of term variables, but this can be enforced again. Hence, we define a special zipping, written $\curlyvee'$, specialized for the let rule, that requires that, if $\Gamma_1 \curlyvee' \Gamma_2$ is defined and if $\exists\alpha$ appears in Γ_2, then $\forall\alpha$ must *not* appear in Γ_1, while, if $\exists\alpha$ appears in Γ_1, then $\forall\alpha$ should also appear in Γ_2, as before. Zipping for the other rules $\curlyvee$ is symmetric and requires that if $\exists\alpha$ appears on one side, then $\forall\alpha$ must not be present on the other side. This restriction permits to reproduce the usage of type variables in System F, while keeping the flexibility of our constructs.

Figure 2.4 on the next page presents the full definition for the zipping operators. They are defined as three-place relations, with the notations $\cdot \curlyvee \cdot = \cdot$ and $\cdot \curlyvee' \cdot = \cdot$, respectively. We use the $=$ sign, to highlight the fact that their third arguments must be understood as outputs, while the others should by interpreted as inputs. Notice, however, that the zipping relations are *not functional*: for example, if $\alpha \neq \beta$, then $\exists\alpha \curlyvee \exists\beta = \exists\alpha, \exists\beta$ holds, as well as $\exists\alpha \curlyvee \exists\beta = \exists\beta, \exists\alpha$. This is not a problem, since Lemma 2.4.12 on page 25 shows that the positions of existential bindings in environments are irrelevant for the different judgments.

We review the main properties of the zipping operators.

Lemma 2.3.1 (Properties of symmetric zipping)**.** *The following assertions hold:*

- Symmetry*: if* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma$*, then* $\Gamma_2 \curlyvee \Gamma_1 = \Gamma$*;*
- Associativity*: if* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma_{12}$ *and* $\Gamma_{12} \curlyvee \Gamma_3 = \Gamma_{123}$*, then there exists* Γ_{23} *such that* $\Gamma_2 \curlyvee \Gamma_3 = \Gamma_{23}$ *and* $\Gamma_1 \curlyvee \Gamma_{23} = \Gamma_{123}$*;*
- Reflexivity in the pure case*: if* $\Gamma\,\mathrm{pure}$*, then* $\Gamma \curlyvee \Gamma = \Gamma$*;*
- *If* $\Gamma_1\,\mathrm{pure}$*, and* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma$*, then* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma_2$*;*

$$
\begin{array}{rcll}
\varepsilon \curlyvee \varepsilon & = & \varepsilon & \\
\Gamma_1, x : \tau \curlyvee \Gamma_2, x : \tau & = & (\Gamma_1 \curlyvee \Gamma_2), x : \tau & \text{if } x \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall \alpha \curlyvee \Gamma_2, \forall \alpha & = & (\Gamma_1 \curlyvee \Gamma_2), \forall \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall (\alpha = \tau) \curlyvee \Gamma_2, \forall (\alpha = \tau) & = & (\Gamma_1 \curlyvee \Gamma_2), \forall (\alpha = \tau) & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha \curlyvee \Gamma_2 & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1 \curlyvee \Gamma_2, \exists \alpha & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2
\end{array}
$$

$$
\begin{array}{rcll}
\varepsilon \barwedge \varepsilon & = & \varepsilon & \\
\Gamma_1, x : \tau \barwedge \Gamma_2, x : \tau & = & (\Gamma_1 \barwedge \Gamma_2), x : \tau & \text{if } x \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall \alpha \barwedge \Gamma_2, \forall \alpha & = & (\Gamma_1 \barwedge \Gamma_2), \forall \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall (\alpha = \tau) \barwedge \Gamma_2, \forall (\alpha = \tau) & = & (\Gamma_1 \barwedge \Gamma_2), \forall (\alpha = \tau) & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha \barwedge \Gamma_2 & = & (\Gamma_1 \barwedge \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1 \barwedge \Gamma_2, \exists \alpha & = & (\Gamma_1 \barwedge \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha \barwedge \Gamma_2, \forall \alpha & = & (\Gamma_1 \barwedge \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2
\end{array}
$$

Figure 2.4: Zipping: definition.

- Distributivity*: if* Γ_1 pure *and* $\Gamma_2 \curlyvee \Gamma_3 = \Gamma_{23}$ *and* $\Gamma_1 \curlyvee \Gamma_{23} = \Gamma_{123}$*, then there exists* Γ_{13} *and* Γ_{12} *such that* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma_{12}$ *and* $\Gamma_1 \curlyvee \Gamma_3 = \Gamma_{13}$ *and* $\Gamma_{12} \curlyvee \Gamma_{13} = \Gamma_{123}$*;*
- *If* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma$*, then* $\Gamma_1 \barwedge \Gamma_2 = \Gamma$*.*

Lemma 2.3.2 (Properties of asymmetric zip)**.** *The following assertions hold:*

- Symmetry*: if* $\operatorname{dom}^{\exists} \Gamma_1 \cap \operatorname{dom}^{\forall} \Gamma_2 = \emptyset$ *and* $\Gamma_1 \barwedge \Gamma_2 = \Gamma$*, then* $\Gamma_2 \barwedge \Gamma_1 = \Gamma$*;*
- Associativity*: if* $\Gamma_1 \barwedge \Gamma_2 = \Gamma_{12}$ *and* $\Gamma_{12} \barwedge \Gamma_3 = \Gamma_{123}$*, then there exists* Γ_{23} *such that* $\Gamma_2 \barwedge \Gamma_3 = \Gamma_{23}$ *and* $\Gamma_1 \barwedge \Gamma_{23} = \Gamma_{123}$*;*
- Reflexivity in the pure case*: if* Γ pure*, then* $\Gamma \barwedge \Gamma = \Gamma$*;*
- *If* Γ_1 pure*, and* $\Gamma_1 \barwedge \Gamma_2 = \Gamma$*, then* $\Gamma_1 \barwedge \Gamma_2 = \Gamma_2$*;*
- *If* Γ_2 pure*, and* $\Gamma_1 \barwedge \Gamma_2 = \Gamma$*, then* $\Gamma_1 \barwedge \Gamma_2 = \Gamma_1$*;*
- Distributivity over the symmetric zip*: if* Γ_1 pure *and* $\Gamma_2 \curlyvee \Gamma_3 = \Gamma_{23}$ *and* $\Gamma_1 \barwedge \Gamma_{23} = \Gamma_{123}$*, then there exists* Γ_{12} *and* Γ_{13} *such that* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma_{12}$ *and* $\Gamma_1 \curlyvee \Gamma_3 = \Gamma_{13}$ *and* $\Gamma_{12} \barwedge \Gamma_{13} = \Gamma_{123}$*;*
- Pseudo-distributivity*: if* Γ_1 pure *and* $\Gamma_2 \barwedge \Gamma_3 = \Gamma_{23}$ *and* $\Gamma_1 \barwedge \Gamma_{23} = \Gamma_{123}$*, then there exists* Γ_1'*,* Γ_{12} *and* Γ_{13} *such that* $\Gamma_1 \curlyvee \Gamma_2 = \Gamma_{12}$ *and* $\Gamma_1' \curlyvee \Gamma_3 = \Gamma_{13}$ *and* $\Gamma_{12} \barwedge \Gamma_{13} = \Gamma_{123}$ *and* $\Gamma_1' \sqsupseteq^{\forall} \Gamma_1$*.*

The relation $\sqsupseteq^{\forall}$ is defined on Figure 2.6 on the next page: the intuition is that $\Gamma_1 \sqsupseteq^{\forall} \Gamma_2$ holds if Γ_1 contains more *universal bindings* than Γ_2. The property of pseudo-distributivity is used to show that the typing judgment is stable under asymmetric zipping, which is required to prove subject reduction: universal weakening is necessary to handle the case of the Let rule.

2.3.2 Syntax

The language $\mathsf{F}^{\curlyvee}$ is based on the explicitly typed version of System F with records and is extended with constructs of Section 2.2.1 on page 8. Types and terms are described in Figure 2.5 on the next page.

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau \to \tau \mid \{(\ell_i : \tau_i)^{i \in 1..n}\} & \textit{(Types)} \\
 & \mid & \forall \alpha.\tau \mid \exists \alpha.\tau & \\
M & ::= & x \mid \lambda(x:\tau)\, M \mid M\ M & \textit{(Terms)} \\
 & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M \mid \Lambda\alpha.\, M \mid M\,\tau & \\
 & \mid & \{(\ell_i = M_i)^{i \in 1..n}\} \mid M.\ell & \\
 & \mid & \exists \alpha.\, M \mid \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\, M \mid (M : \tau) & \\
 & \mid & \mathsf{open}\ \langle\alpha\rangle\ M \mid \nu\alpha.\, M & \\
v & ::= & \lambda(x:\tau)\, M \mid \Lambda\alpha.\, M & \textit{(Values)} \\
 & \mid & \{(\ell_i = v_i)^{i \in 1..n}\} \mid \exists\beta.\, \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\, v & \\
 & \mid & (v : \tau) \quad \text{if } v \text{ is not a coercion} & \\
w & ::= & v \mid \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\, w & \textit{(Results)}
\end{array}
$$

Figure 2.5: Syntax: types, terms, values, and results.

$$
\frac{}{\varepsilon \sqsupseteq^{\forall} \varepsilon}
\qquad
\frac{\Gamma \sqsupseteq^{\forall} \Gamma' \qquad x \notin \operatorname{dom}\Gamma}{\Gamma, x:\tau \sqsupseteq^{\forall} \Gamma', x:\tau}
\qquad
\frac{\Gamma \sqsupseteq^{\forall} \Gamma' \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \forall\alpha \sqsupseteq^{\forall} \Gamma', \forall\alpha}
\qquad
\frac{\Gamma \sqsupseteq^{\forall} \Gamma' \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \exists\alpha \sqsupseteq^{\forall} \Gamma', \exists\alpha}
$$

$$
\frac{\Gamma \sqsupseteq^{\forall} \Gamma' \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \forall(\alpha = \tau) \sqsupseteq^{\forall} \Gamma', \forall(\alpha = \tau)}
\qquad
\frac{\Gamma \sqsupseteq^{\forall} \Gamma' \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \forall\alpha \sqsupseteq^{\forall} \Gamma'}
$$

Figure 2.6: Universally-weakened environment.

$$
\textsc{Wf-Var}\ \frac{\Gamma \vdash \mathrm{ok} \qquad \alpha \in \operatorname{dom}^{\exists}\Gamma \cup \operatorname{dom}^{\forall}\Gamma \cup \operatorname{dom}^{\simeq}\Gamma}{\Gamma \vdash \alpha :: \star}
\qquad
\textsc{Wf-Arrow}\ \frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma \vdash \tau_2 :: \star}{\Gamma \vdash \tau_1 \to \tau_2 :: \star}
\qquad
\textsc{Wf-RecordEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} :: \star}
$$

$$
\textsc{Wf-Record}\ \frac{\Gamma \vdash \tau :: \star \qquad \forall i \in 1..n, \ell \neq \ell_i \qquad \Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} :: \star}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} :: \star}
\qquad
\textsc{Wf-Forall}\ \frac{\Gamma, \forall\alpha \vdash \tau :: \star}{\Gamma \vdash \forall\alpha.\tau :: \star}
\qquad
\textsc{Wf-Exists}\ \frac{\Gamma, \forall\alpha \vdash \tau :: \star}{\Gamma \vdash \exists\alpha.\tau :: \star}
$$

Figure 2.7: Wellformedness of types.

$$
\textsc{Ok-Empty}\ \frac{}{\varepsilon \vdash \mathrm{ok}}
\qquad
\textsc{Ok-Var}\ \frac{\Gamma \vdash \tau :: \star \qquad x \notin \operatorname{dom}\Gamma}{\Gamma, x:\tau \vdash \mathrm{ok}}
\qquad
\textsc{Ok-Exists}\ \frac{\Gamma \vdash \mathrm{ok} \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \exists\alpha \vdash \mathrm{ok}}
\qquad
\textsc{Ok-Forall}\ \frac{\Gamma \vdash \mathrm{ok} \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \forall\alpha \vdash \mathrm{ok}}
\qquad
\textsc{Ok-Eq}\ \frac{\Gamma \vdash \tau :: \star \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \forall(\alpha = \tau) \vdash \mathrm{ok}}
$$

Figure 2.8: Wellformedness of environments.

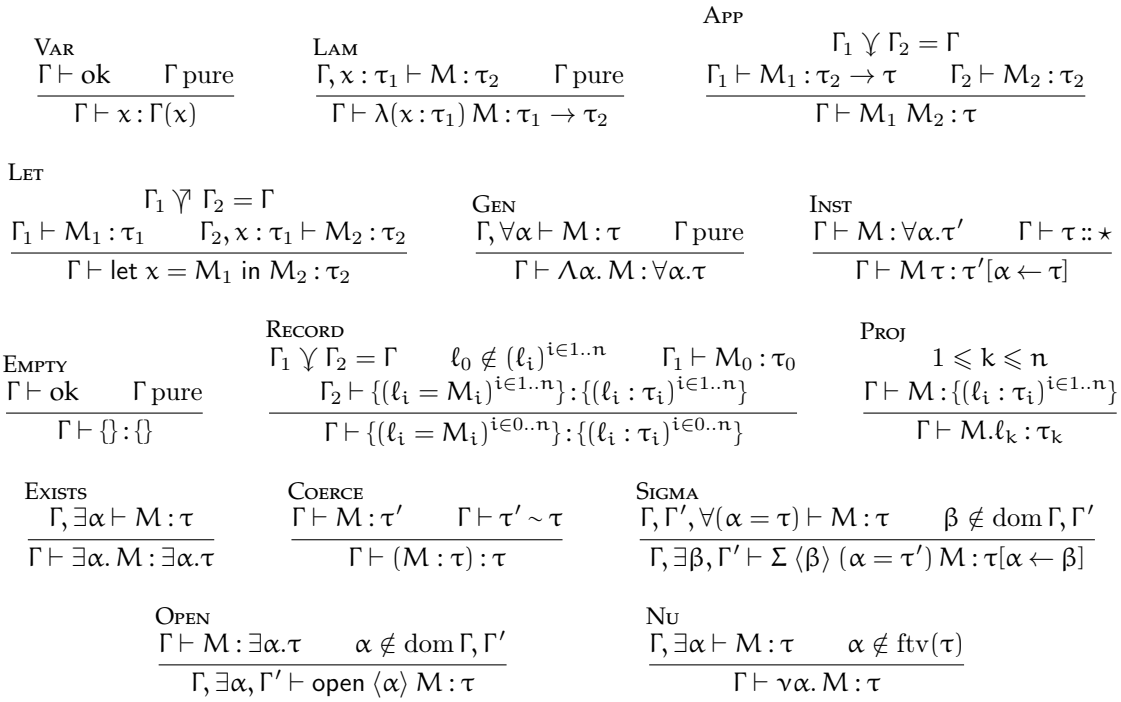


Figure 2.9: Typing rules of the core system.

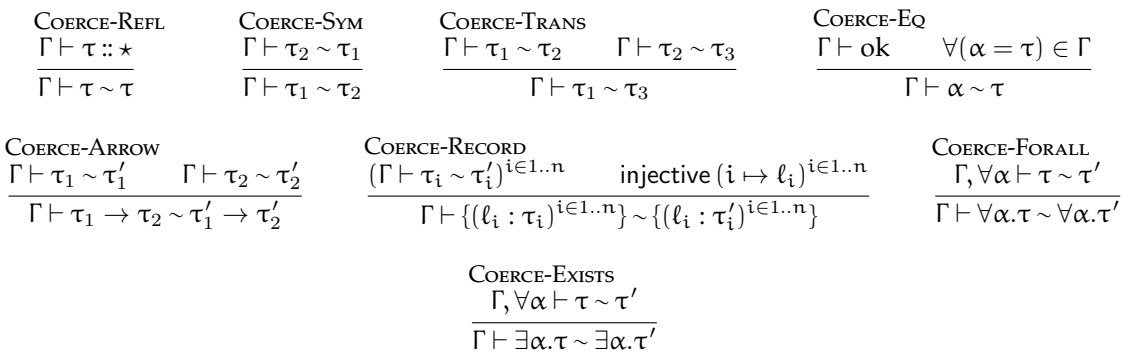


Figure 2.10: Coercible types.

As open existentials do not introduce new forms of types, types of $\mathsf{F}^{\curlyvee}$ are type variables, arrow types, record types, universal types, and existential types. The notation $(\ell_i : \tau_i)^{i\in 1..n}$ stands for a sequence of n pairs, each composed of a label and a type. Type wellformedness is defined as usual, *i.e.* as in Figure 2.23 on page 45 without the rule for recursive types: environments contain all type variables used in types.

Terms of $\mathsf{F}^{\curlyvee}$ are variables, functions (whose arguments are explicitly typed), applications, let-bindings, type generalizations and applications, introductions and projections of records, and the five constructs for open existentials described above: existential introductions, open witness definitions, coercions, openings, and restrictions. Record fields are pairs $\ell = M$ of a label name ℓ and a term M. The label name is used to access the field externally, as usual with records.

We write $\mathrm{ftv}(\tau)$ (respectively $\mathrm{ftv}(M)$) to denote the set of free type variables of a type τ (respectively a term M).

2.3.3 Typing rules

Typing rules for open existentials have already been presented in Section 2.2.1 on page 8. The remaining typing rules (Figure 2.9 on the facing page) are as in System F with two small differences: first, as mentioned above, typing rules with several typing judgments as premises use zipping instead of equality to relate their typing environments. This is the case of rules App, Let, and Record. Second, typing rules must also ensure that values can be substituted without breaking linearity, which is the case when the typing environment does not contain existential items, *i.e.* when the environment is *pure*. This condition appears as an additional premise of typing rules of expressions that are also values (namely, rules Var, Lam, Gen, and Empty). Purity will be used and explained in more details in Section 2.3.4.

Because rule Open makes the environment decrease (if it is read bottom-up), the property of weakening is *not* provable in its whole generality: one can only weaken a judgment by a non-linear item that does not depend on linear items. This is sufficient for the proof of soundness. A primitive weakening rule will be added when considering extensions of core $\mathsf{F}^{\curlyvee}$ (Section 2.6.1 on page 36).

Notice also that the rule Let cannot be derived from Lam and App because of the purity condition that rule Lam requires.

Our typing rules ensure that environments and types are wellformed. Wellformedness judgments for types and environments are defined in Figure 2.7 and Figure 2.7 on page 17. The judgments enforce that the free variables of types are included in the domains of environments, and that environments contain distinct variables, and that every type expression that occurs in an environment is wellformed.

2.3.4 Reduction semantics

The language $\mathsf{F}^{\curlyvee}$ is equipped with a small-step call-by-value reduction semantics. We begin with important remarks about substitutability, then define and explain values, and finally describe the reduction steps.

Substitution and purity

Some terms *cannot* be safely substituted, since substitution could violate the linear treatment of openings and open witness definitions. It turns out that *pure* terms, *i.e.* terms that are typable in a pure environment, behave well with respect to substitution:

Lemma 2.3.3 (Substitution lemma)**.** *Assume that* $\Gamma \vdash M : \tau$ *and* $\Gamma', x : \tau, \Gamma'' \vdash M' : \tau'$ *hold, where* Γ *is pure and* $\Gamma \curlyvee \Gamma'$ *is well defined. Then* $(\Gamma \curlyvee \Gamma'), \Gamma'' \vdash M'[x \leftarrow M] : \tau'$ *also holds.*

$$
\begin{aligned}
&\quad \mathsf{let}\ x = \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\ (1 : \alpha)\ \mathsf{in}\ \{\ell_1 = x\ ;\ \ell_2 = (\lambda(y : \beta)\, y)\ x\} \\
&\longrightarrow \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int}) \\
&\qquad \mathsf{let}\ x = (1 : \alpha)\ \mathsf{in}\ \{\ell_1 = x\ ;\ \ell_2 = (\lambda(y : \alpha)\, y)\ x\} \\
&\longrightarrow \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\, \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = (\lambda(y : \alpha)\, y)\ (1 : \alpha)\} \\
&\longrightarrow \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\, \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = (1 : \alpha)\}
\end{aligned}
$$

Figure 2.11: Example of extrusion.

Therefore, values are substitutable if we restrict them to pure terms. But conversely, every irreducible term is *not necessarily* a pure term.

Results and values

Results are well-behaved irreducible terms. Results include values. In System F (as in many other languages) results actually coincide with values. However, this need not be the case. In $\mathsf{F}^{\forall}$, results also include terms such as $\Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, \lambda(x : \alpha)\, x$, which are well-behaved and cannot be further reduced, but are not values, as they are not pure and thus not substitutable.

Therefore we have to distinguish *results*, which are irreducible terms, from *values*, which are *pure* results.

More precisely, values are defined in Figure 2.5 on page 17. They are either variables, functions, generalizations, records of values, existential values, or coerced values. Note that nested coercions are not values—they must be further reduced. Note also that no evaluation takes place under λs or Λs. Finally, results are values preceded by a (possibly empty) sequence of Σs.

The purity premises in some of the typing rules ensure that values are pure, hence, by Lemma 2.3.3 on the previous page, substitutable.

Lemma 2.3.4 (Purity of values)**.** *If* $\Gamma \vdash v : \tau$ *holds, then* Γ *is pure.*

Extrusions

Values are substitutable, but some results are not values: for instance, a value that is prefixed by a non-empty sequence of Σs is a result, but not a value. How can we handle these results, when they ought to be substituted, without breaking linearity? Our solution is to extrude the Σs *just enough* to expose and perform the next reduction step. Notice that we could also have chosen to aggressively extrude Σs, and even extrude νs as much as possible, but then, it would have meant that we removed abstraction from the beginning, before computation is performed. Our solution, in constrast, reveals witnesses only when it is necessary (abstraction is kept as long as possible), and keeps local abstract types local (since νs are not extruded at all).

For example, consider the reduction steps in Figure 2.11. The initial expression is a let-binding of the form $\mathsf{let}\ x = w\ \mathsf{in}\ M$ where w is the result form $\Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\, (1 : \alpha)$. Hence, the next expected reduction step is the substitution of w for x in M. However, since x occurs twice in M, this would duplicate the (open witness) definition appearing in w, thus breaking the linear use of β. The solution is to first *extrude* the Σ binding outside of the let-binding, so that the expression bound to x becomes the substitutable value form $(1 : \alpha)$. However, by enlarging the scope of Σ, we have put M in its scope, in which the external name β occurs. Therefore, we replace it with the internal one in the enlarged scope. Then, we may perform let-reduction safely and further reduce the redex that has been created.

In this particular example, it is not *unsafe* to duplicate the open witness definition, since the duplicated occurences of Σ would remain *consistent*: they would all define the same witness. Dupli-

cating Σs, however, does not respect the linear discipline we intend to impose. Leveraging linearity is discussed in Section 2.8.1 on page 52.

In order to maintain linearity, the reduction semantics will be set so that Σs can always be extruded out of redex forms. Note that the separation of witness definitions from coercions (*i.e.* splitting pack) is essential here: if the two constructs were bound together, coercions should be necessarily extruded too, which would be hard to achieve in a local manner. Here, only the witness definitions are extruded, while the coercions simply stay where they are.

Openings also introduce linear items into the environment and thus preclude substitution. Note however that they are neither part of values nor of results, because they can be eliminated: by reduction, an opening $\mathsf{open}\ \langle\beta\rangle\ M$ will eventually lead to an "open-exists" pattern $\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\, M'$. This combination just performs a transfer of an existential resource from the inner name α to the outer one β, as demonstrated by the following derivation:

$$\textsc{Open}\ \dfrac{\textsc{Exists}\ \dfrac{\Gamma, \exists\alpha \vdash M : \tau}{\Gamma \vdash \exists\alpha.\, M : \exists\alpha.\tau}}{\Gamma, \exists\beta \vdash \mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\, M : \tau[\alpha \leftarrow \beta]}$$

Therefore, the pattern $\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\, M$ can simply be eliminated into a renaming from the internal to the external name $M[\alpha \leftarrow \beta]$. This way, reduction makes the bottom-left cycle of Figure 2.1 on page 13 vanish.

Reduction

The semantics of $\mathsf{F}^{\forall}$ is given by a call-by-value reduction strategy, described by a small-step reduction relation, that does not rely on types (it is compatible with type erasure). We fix a left-to-right evaluation order so that the semantics is deterministic, although we could have left the order unspecified. By contrast, having a call-by-value strategy and a weak-reduction is essential: the call-by-value strategy ensures that the linear discipline is preserved. The problem of strong reduction is discussed in Section 2.8.1 on page 52.

The notation $M_1 \longrightarrow M_2$ denotes that the term M_1 reduces to the term M_2 in some evaluation context. Evaluation contexts are described in Figure 2.12 on the following page. Note that, as opposed to RTG [Dre07a], evaluation also takes place under existential bindings. A one-step reduction is the application of a reduction rule in some evaluation context. The reduction relation is the transitive closure of the one-step reduction relation. Reduction steps are sorted into three groups.

Rules of the main group describe the contraction of redexes. The let-reduction, the β-reduction, the reduction of type applications, and the record projection are as usual. The next rule of this group is the reduction of the "open-exists" pattern explained above. Notice that type substitution is a partial function on terms, because syntax is not stable under type substitution: for instance, $(\mathsf{open}\ \langle\beta\rangle\ M)[\beta \leftarrow \tau]$ is undefined: by contrast, renaming is a total function. The type system ensures that type substitution is only performed when it is well-defined (see Lemma 2.4.9 and Lemma 2.4.10 on page 25). The last rule of this group is responsible for the erasure of restricted open witness definitions: it replaces the type variable of a witness with the witness itself: the same substitution occurs in System F while unpacking an existential package.

The second group of rules implements the extrusion of Σs through every other construct: more precisely, extrusion proceeds through any evaluation context. Rules Sigma-Exists and Sigma-Nu have an extra condition to forbid extrusion of Σs whose witnesses depend on a variable that is bound by the context. Rule Sigma-Sigma substitutes the definition of the outer one to delete dependencies: this allows to swap two Σs, even if one syntactically depends on the other.

Finally, the third group of reduction rules keeps track of coercions during reduction, as exemplified by rule Coerce-App. Notice that nested coercions are merged, the outer one taking priority (Rule Coerce-Coerce), which makes the top-most cycle of Figure 2.1 on page 13 vanish.

$$
\begin{array}{rll}
\mathsf{let}\ x = v\ \mathsf{in}\ M & \longrightarrow M[x \leftarrow v] & \textsc{Redex-Let} \\
(\lambda(x:\tau)\,M)\,v & \longrightarrow \mathsf{let}\ x = v\ \mathsf{in}\ M & \textsc{Redex-App} \\
(\Lambda\alpha.\,M)\,\tau & \longrightarrow M[\alpha \leftarrow \tau] & \textsc{Redex-Inst} \\
\{(\ell_i = v_i)^{i\in 1..n}\}.\ell_k & \longrightarrow v_k \quad \text{if } 1 \leqslant k \leqslant n & \textsc{Redex-Proj} \\
\mathsf{open}\,\langle\beta\rangle\,\exists\alpha.\,w & \longrightarrow w[\alpha \leftarrow \beta] & \textsc{Redex-Open} \\
\nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w & \longrightarrow w[\alpha \leftarrow \tau] \quad \text{if } \beta \notin \mathrm{ftv}(\tau) \cup \mathrm{ftv}(w) & \textsc{Redex-Nu} \\
\\
\mathsf{let}\ x = \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w\ \mathsf{in}\ M & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau) & \textsc{Sigma-Let} \\
 & \quad \mathsf{let}\ x = w\ \mathsf{in}\ M[\beta \leftarrow \alpha] \quad \text{if } \alpha \notin \mathrm{ftv}(M) & \\
(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w_1)\,w_2 & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(w_1\,w_2[\beta \leftarrow \alpha]) \quad \text{if } \alpha \notin \mathrm{ftv}(w_2) & \textsc{Sigma-App1} \\
w_1\,(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w_2) & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(w_1[\beta \leftarrow \alpha]\,w_2) \quad \text{if } \alpha \notin \mathrm{ftv}(w_1) & \textsc{Sigma-App2} \\
(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w)\,\tau' & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(w\,\tau'[\beta \leftarrow \alpha]) \quad \text{if } \alpha \notin \mathrm{ftv}(\tau') & \textsc{Sigma-Inst} \\
(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w).\ell & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(w.\ell) & \textsc{Sigma-Proj} \\
\{(\ell_i = w_i)^{i\in 1..n}\ ; & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\{(\ell_i = w_i[\beta \leftarrow \alpha])^{i\in 1..n}\ ; & \textsc{Sigma-Record} \\
\quad \ell = \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w\ ; & \qquad\qquad \ell\ = w\ ; & \\
\quad (\ell'_j = w'_j)^{j\in 1..m}\} & \qquad\qquad (\ell'_j = w'_j[\beta \leftarrow \alpha])^{j\in 1..m}\} & \\
 & \quad \text{if } \alpha \notin (\mathrm{ftv}(w_i))^{i\in 1..n} \cup (\mathrm{ftv}(w'_j))^{j\in 1..m} & \\
\mathsf{open}\,\langle\gamma\rangle\,(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w) & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(\mathsf{open}\,\langle\gamma\rangle\,w) \quad \text{if } \gamma \notin \{\alpha, \beta\} & \textsc{Sigma-Open} \\
\exists\gamma.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\exists\gamma.\,w \quad \text{if } \gamma \notin \{\alpha, \beta\} \cup \mathrm{ftv}(\tau) & \textsc{Sigma-Exists} \\
\nu\gamma.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\nu\gamma.\,w \quad \text{if } \gamma \notin \{\alpha, \beta\} \cup \mathrm{ftv}(\tau) & \textsc{Sigma-Nu} \\
(\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w : \tau') & \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(w : \tau'[\beta \leftarrow \alpha]) \quad \text{if } \alpha \notin \mathrm{ftv}(\tau') & \textsc{Sigma-Coerce} \\
\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau_1) & \longrightarrow \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \tau_2[\alpha_1 \leftarrow \tau_1]) & \textsc{Sigma-Sigma} \\
\Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \tau_2)\,w & \quad \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau_1[\beta_2 \leftarrow \alpha_2])\,w & \\
\\
((\lambda(x:\tau_0)\,M) : \tau_1 \to \tau_2)\,v & \longrightarrow ((\lambda(x:\tau_0)\,M)\,(v:\tau_0) : \tau_2) & \textsc{Coerce-App} \\
(u : \forall\alpha.\tau')\,\tau & \longrightarrow (u\,\tau : \tau'[\alpha \leftarrow \tau]) & \textsc{Coerce-Inst} \\
(u : \{(\ell_i : \tau_i)^{i\in 1..n}\}).\ell_k & \longrightarrow (u.\ell_k : \tau_k) \quad \text{if } 1 \leqslant k \leqslant n & \textsc{Coerce-Proj} \\
\mathsf{open}\,\langle\alpha\rangle\,(u : \exists\alpha.\tau) & \longrightarrow (\mathsf{open}\,\langle\alpha\rangle\,u : \tau) & \textsc{Coerce-Open} \\
((u:\tau):\tau') & \longrightarrow (u:\tau') & \textsc{Coerce-Coerce}
\end{array}
$$

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid E\,M \mid w\,E \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid E\,\tau \\
 & \mid & \{(\ell_i = w_i)^{i\in 1..k}\ ;\ \ell_{k+1} = E\ ;\ (\ell_i = M_i)^{i\in k+2..n}\} \mid E.\ell \\
 & \mid & \exists\alpha.\,E \mid \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,E \mid (E:\tau) \mid \mathsf{open}\,\langle\alpha\rangle\,E \mid \nu\alpha.\,E
\end{array}
\qquad
\textsc{Context}\ \frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}
$$

Figure 2.12: Reduction rules.

Remark that only Σs are extruded: every local introduction of resources by a ν stays local and is eventually eliminated. Coercions are not extruded either. Notice that the rule Sigma-Sigma can be applied infinitely many times: this might lead to stuttering. This is taken into account in the statement of *progress* (Proposition 2.4.24 on page 28), so that its proof does not take advantage of stuttering.

2.4 Soundness

We show the soundness of the type system in a syntactic manner, based on the subject reduction and progress lemmas. Moreover, since the reduction can trivially loop, due to the successive application of Sigma-Sigma, we will show an enhanced version of the progress lemma. The soundness proof was implemented and verified in the Coq proof assistant for a large subset of Core $F^{\forall}$. Details on the mechanized proof can be found in Section 2.4.6 on page 28.

2.4.1 Basic syntactic lemmas

Every context, term or types appearing in any judgment is wellformed:

Lemma 2.4.1 (Regularity)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \mathrm{ok}$ *and* $x : \tau \in \Gamma$ *or* $\forall(\alpha = \tau) \in \Gamma$*, then* $\Gamma \vdash \tau :: \star$*;*
- *If* $\Gamma \vdash \tau :: \star$*, then* $\Gamma \vdash \mathrm{ok}$*;*
- *If* $\Gamma \vdash \tau \sim \tau'$*, then* $\Gamma \vdash \tau :: \star$ *and* $\Gamma \vdash \tau' :: \star$*;*
- *If* $\Gamma \vdash M : \tau$*, then* $\Gamma \vdash \tau :: \star$*.*

For any typing judgment, every free variable has been introduced in the context.

Lemma 2.4.2 (Free variables)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \star$*, then* $\mathrm{ftv}(\tau) \subseteq \mathrm{dom}\,\Gamma \setminus \mathrm{dom}^{var}\,\Gamma$*;*
- *If* $\Gamma \vdash M : \tau$*, then* $\mathrm{ftv}(M) \subseteq \mathrm{dom}\,\Gamma \setminus \mathrm{dom}^{var}\,\Gamma$ *and* $\mathrm{fv}(M) \subseteq \mathrm{dom}^{var}\,\Gamma$*.*

Every judgment is stable under renamings for a sufficiently fresh variable:

Lemma 2.4.3 (Stability under renaming of term variables)**.** *Assume* $y \notin \mathrm{dom}\,\Gamma_1, \Gamma_2$*. Then, the following assertions hold:*

- *If* $\Gamma_1, x : \tau, \Gamma_2 \vdash \mathrm{ok}$*, then* $\Gamma_1, y : \tau, \Gamma_2 \vdash \mathrm{ok}$*;*
- *If* $\Gamma_1, x : \tau, \Gamma_2 \vdash \tau' :: \star$*, then* $\Gamma_1, y : \tau, \Gamma_2 \vdash \tau' :: \star$*;*
- *If* $\Gamma_1, x : \tau, \Gamma_2 \vdash \tau' \sim \tau''$*, then* $\Gamma_1, y : \tau, \Gamma_2 \vdash \tau' \sim \tau''$*;*
- *If* $\Gamma_1, x : \tau, \Gamma_2 \vdash M' : \tau'$*, then* $\Gamma_1, y : \tau, \Gamma_2 \vdash M'[x \leftarrow y] : \tau'$*.*

Lemma 2.4.4 (Stability under renaming of type variables)**.** *Assume* $\beta \notin \mathrm{dom}\,\Gamma_1, \Gamma_2$*. Then, the following assertions hold:*

- *If* $\Gamma_1, \forall\alpha, \Gamma_2 \vdash \mathrm{ok}$*, then* $\Gamma_1, \forall\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \mathrm{ok}$*;*
- *If* $\Gamma_1, \forall\alpha, \Gamma_2 \vdash \tau' :: \star$*, then* $\Gamma_1, \forall\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] :: \star$*;*
- *If* $\Gamma_1, \forall\alpha, \Gamma_2 \vdash \tau' \sim \tau''$*, then* $\Gamma_1, \forall\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] \sim \tau''[\alpha \leftarrow \beta]$*;*

- *If* $\Gamma_1, \forall\alpha, \Gamma_2 \vdash M' : \tau'$, *then* $\Gamma_1, \forall\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash M'[\alpha \leftarrow \beta] : \tau'[\alpha \leftarrow \beta]$;
- *If* $\Gamma_1, \exists\alpha, \Gamma_2 \vdash \mathrm{ok}$, *then* $\Gamma_1, \exists\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \mathrm{ok}$;
- *If* $\Gamma_1, \exists\alpha, \Gamma_2 \vdash \tau' :: \star$, *then* $\Gamma_1, \exists\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] :: \star$;
- *If* $\Gamma_1, \exists\alpha, \Gamma_2 \vdash \tau' \sim \tau''$, *then* $\Gamma_1, \exists\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] \sim \tau''[\alpha \leftarrow \beta]$;
- *If* $\Gamma_1, \exists\alpha, \Gamma_2 \vdash M' : \tau'$, *then* $\Gamma_1, \exists\beta, \Gamma_2[\alpha \leftarrow \beta] \vdash M'[\alpha \leftarrow \beta] : \tau'[\alpha \leftarrow \beta]$;
- *If* $\Gamma_1, \forall(\alpha = \tau), \Gamma_2 \vdash \mathrm{ok}$, *then* $\Gamma_1, \forall(\beta = \tau), \Gamma_2[\alpha \leftarrow \beta] \vdash \mathrm{ok}$;
- *If* $\Gamma_1, \forall(\alpha = \tau), \Gamma_2 \vdash \tau' :: \star$, *then* $\Gamma_1, \forall(\beta = \tau), \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] :: \star$;
- *If* $\Gamma_1, \forall(\alpha = \tau), \Gamma_2 \vdash \tau' \sim \tau''$, *then* $\Gamma_1, \forall(\beta = \tau), \Gamma_2[\alpha \leftarrow \beta] \vdash \tau'[\alpha \leftarrow \beta] \sim \tau''[\alpha \leftarrow \beta]$;
- *If* $\Gamma_1, \forall(\alpha = \tau), \Gamma_2 \vdash M' : \tau'$, *then* $\Gamma_1, \forall(\beta = \tau), \Gamma_2[\alpha \leftarrow \beta] \vdash M'[\alpha \leftarrow \beta] : \tau'[\alpha \leftarrow \beta]$.

The set of free variables decreases with reduction. Reduction is stable under substitution of terms and renaming of type variables.

Lemma 2.4.5. *Assume that* $M \longrightarrow^\star M'$. *Then, the following assertions hold:*

- $\mathrm{fv}(M') \subseteq \mathrm{fv}(M)$ *and* $\mathrm{ftv}(M') \subseteq \mathrm{ftv}(M)$;
- $M[x \leftarrow N] \longrightarrow^\star M'[x \leftarrow N]$;
- *If* $\beta \notin \mathrm{ftv}(M)$, *then* $M[\alpha \leftarrow \beta] \longrightarrow^\star M'[\alpha \leftarrow \beta]$.

The freshness condition on the last item is necessary, because reduction is guarded by conditions on type variables, as in Redex-Nu and Sigma-Open for instance. To prove the last item, the first item (the set of free variables decreases with reduction) is required, so that the induction in the transitive case goes through.

2.4.2 Main syntactic lemmas

In this section, we review the main lemmas about the typing judgment, that are used in the proof of the soundness properties. An interesting property with respect to weakening is the absence of existential dependency: nothing in the environment can depend on existential bindings.

Lemma 2.4.6 (Absence of existential dependency)**.** *If* $\Gamma \vdash M : \tau$ *and* $x : \tau' \in \Gamma$ *or* $\forall(\alpha = \tau') \in \Gamma$, *then* $\mathrm{ftv}(\tau') \cap \mathrm{dom}^\exists\,\Gamma = \emptyset$.

The lemma is valid thanks to the rules Open and Sigma: they indeed remove an existential binding $\exists\beta$, when read bottom-up, and because the smaller environment must be wellformed, β must not appear in any ot its bindings.

The weakening lemma follows. Notice that it is restricted, as it was already mentioned, to weakening with contexts that do not depend on existential bindings. This lemma needs a strengthened version so that induction goes through, but we omit it here.

Lemma 2.4.7 (Weakening)**.** *Assume* $\Gamma, \Gamma' \vdash \mathrm{ok}$. *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \star$, *then* $\Gamma, \Gamma' \vdash \tau :: \star$;
- *If* $\Gamma \vdash \tau \sim \tau'$, *then* $\Gamma, \Gamma' \vdash \tau \sim \tau'$;
- *If* $\Gamma \vdash M : \tau$ *and* $\mathrm{ftv}(\Gamma') \cap \mathrm{dom}^\exists\,\Gamma = \emptyset$, *then* $\Gamma, \Gamma' \vdash M : \tau$.

It is necessary to prove the universal weakening, because of the pseudo-distributivity of the zipping operator (Lemma 2.3.2 on page 16). Note that it is not a direct consequence of the previous weakening lemma: indeed, Lemma 2.4.7 allows to add several bindings at a single place of the environment, while universal weakening permits to add universal bindings *anywhere* in the environment.

Lemma 2.4.8 (Universal weakening). *Assume* $\Gamma' \sqsupseteq^{\forall} \Gamma$. *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \star$, *then* $\Gamma' \vdash \tau :: \star$;
- *If* $\Gamma \vdash \tau \sim \tau'$, *then* $\Gamma' \vdash \tau \sim \tau'$;
- *If* $\Gamma \vdash M : \tau$, *then* $\Gamma' \vdash M : \tau$.

The instantiation lemmas follow. The first one ensures that one can replace a universal binding by a well-formed equation: in systems with subtyping, this is analogous to *narrowing*. The second lemma permits to unfold an equation everywhere. The usual instantiation lemma results from the combination of the two.

Lemma 2.4.9 (Instantiation by equation). *Assume that* $\Gamma \vdash \tau :: \star$ *and* $\Gamma, \forall\alpha, \Gamma' \vdash M : \tau'$ *hold and that no free type variable of* τ *is existentially bound in* Γ. *Then* $\Gamma, \forall(\alpha = \tau), \Gamma' \vdash M : \tau'$ *holds.*

Lemma 2.4.10 (Instantiation by substitution). *Assume that* $\Gamma, \forall(\alpha = \tau), \Gamma' \vdash M : \tau'$ *holds. Then* $M[\alpha \leftarrow \tau]$ *is well-defined and* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash M[\alpha \leftarrow \tau] : \tau'[\alpha \leftarrow \tau]$ *holds.*

The substitution lemma states that substitution is allowed for pure terms, that are wellformed in an environment for which zipping is well defined.

Lemma 2.4.11 (Substitution). *Assume that* $\Gamma_1 \vdash M_1 : \tau_1$ *and* $\Gamma_1', x : \tau_1, \Gamma_2 \vdash M_2 : \tau_2$ *and* $\Gamma_3 = \Gamma_1 \curlyvee \Gamma_1'$. *Then* $\Gamma_3, \Gamma_2 \vdash M_2[x \leftarrow M_1] : \tau_2$ *holds.*

Proof. By induction on the typing derivation for M_2, with the use of Lemma 2.4.7 on the facing page, Lemma 2.3.2 on page 16 and Lemma 2.4.8. □

The weakening, substitution and instantiation lemmas are usually sufficient to prove type preservation, but not in $F^{\forall}$, because extrusions induce swapping of bindings in environments. The following lemma contains the extra results required by the extrusion rules.

Lemma 2.4.12 (Swappings in environments). *The following assertions hold:*

- *If* $\Gamma_1, \Gamma_2, \forall\alpha, \Gamma_3 \vdash M : \tau$, *then* $\Gamma_1, \forall\alpha, \Gamma_2, \Gamma_3 \vdash M : \tau$;
- *If* $\Gamma_1, \Gamma_2, \exists\alpha, \Gamma_3 \vdash M : \tau$, *then* $\Gamma_1, \exists\alpha, \Gamma_2, \Gamma_3 \vdash M : \tau$;
- *If* $\Gamma_1, \forall\alpha, \Gamma_2, \Gamma_3 \vdash M : \tau$ *and* $\alpha \notin \mathrm{ftv}(\Gamma_2)$, *then* $\Gamma_1, \Gamma_2, \forall\alpha, \Gamma_3 \vdash M : \tau$;
- *If* $\Gamma_1, \exists\alpha, \Gamma_2, \Gamma_3 \vdash M : \tau$ *and* $\alpha \notin \mathrm{ftv}(\Gamma_2)$, *then* $\Gamma_1, \Gamma_2, \exists\alpha, \Gamma_3 \vdash M : \tau$;
- *If* $\Gamma_1, \forall(\alpha_1 = \tau_1), \forall(\alpha_2 = \tau_2), \Gamma_2 \vdash M : \tau$, *then* $\Gamma_1, \forall(\alpha_2 = \tau_2[\alpha_1 \leftarrow \tau_1]), \forall(\alpha_1 = \tau_1), \Gamma_2 \vdash M : \tau$.

2.4.3 Properties of coercibility

The reduction rules that deal with coercions require results on the coercibility relation.

Lemma 2.4.13 (Basic properties of coercibility). *The following assertions hold:*

Reflexivity: *if* $\Gamma \vdash \tau :: \star$, *then* $\Gamma \vdash \tau \sim \tau$;

Unfolding of an equation: *if* $\Gamma \vdash \tau :: \star$ *and* $\forall(\alpha = \tau') \in \Gamma$, *then* $\Gamma \vdash \tau \sim \tau[\alpha \leftarrow \tau']$;

$$\frac{\forall \alpha \in \Gamma \vee \exists \alpha \in \Gamma \qquad \Gamma \vdash \mathrm{ok}}{\Gamma \vdash \alpha \Rrightarrow \alpha} \qquad \frac{\forall (\alpha = \tau) \in \Gamma \qquad \Gamma \vdash \tau \Rrightarrow \tau'}{\Gamma \vdash \alpha \Rrightarrow \tau'} \qquad \frac{\Gamma \vdash \tau_1 \Rrightarrow \tau_1' \qquad \Gamma \vdash \tau_2 \Rrightarrow \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \Rrightarrow \tau_1' \to \tau_2'}$$

$$\frac{\text{for every } i \in 1..n, \Gamma \vdash \tau_i \Rrightarrow \tau_i'}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \Rrightarrow \{(\ell_i : \tau_i')^{i \in 1..n}\}} \qquad \frac{\Gamma, \forall \alpha \vdash \tau \Rrightarrow \tau'}{\Gamma \vdash \forall \alpha.\tau \Rrightarrow \forall \alpha.\tau'} \qquad \frac{\Gamma, \forall \alpha \vdash \tau \Rrightarrow \tau'}{\Gamma \vdash \exists \alpha.\tau \Rrightarrow \exists \alpha.\tau'}$$

Figure 2.13: Type normalization.

Stability under equivalent assumptions: $\Gamma_1, \forall(\alpha = \tau_1), \Gamma_2 \vdash \tau \sim \tau'$ *and* $\Gamma_1 \vdash \tau_1 \sim \tau_1'$, *then* $\Gamma_1, \forall(\alpha = \tau_1'), \Gamma_2 \vdash \tau \sim \tau'$.

Inversion and consistency lemmas, defined hereafter (see Lemma 2.4.19 and Lemma 2.4.20), cannot be directly proved by induction on the judgment of type equality, due to the rule of transitivity. To proceed, we first define an algorithmic version of coercibility, which is syntax directed, hence easy to analyze, and prove it equivalent with the declarative version.

Definition 2.4.1 (Type normalization, algorithmic coercibility)**.** Type normalization, denoted by $\Gamma \vdash \tau \Rrightarrow \tau'$ is defined in Figure 2.13. Algorithmic coercibility, denoted by $\Gamma \vdash \tau_1 \Longleftrightarrow \tau_2$, holds iff there exists τ' such that $\Gamma \vdash \tau_1 \Rrightarrow \tau'$ and $\Gamma \vdash \tau_2 \Rrightarrow \tau'$.

Type normalization enjoys the following properties: it is a function that is total on well defined inputs. It is moreover idempotent, and also correct with respect to coercibility.

Lemma 2.4.14 (Determinacy of normalization)**.** *If* $\Gamma \vdash \tau \Rrightarrow \tau_1$ *and* $\Gamma \vdash \tau \Rrightarrow \tau_2$, *then* $\tau_1 = \tau_2$;

Lemma 2.4.15 (Idempotency of normalization)**.** *If* $\Gamma \vdash \tau \Rrightarrow \tau_1$ *and* $\Gamma \vdash \tau_1 \Rrightarrow \tau_2$, *then* $\tau_1 = \tau_2$.

Lemma 2.4.16 (Productivity of normalization)**.** *If* $\Gamma \vdash \tau : \star$, *then there exists* τ' *such that* $\Gamma \vdash \tau \Rrightarrow \tau'$.

Lemma 2.4.17 (Correctness of normalization)**.** *If* $\Gamma \vdash \tau \Rrightarrow \tau'$, *then* $\Gamma \vdash \tau \sim \tau'$.

Lemma 2.4.18 (Adequacy of algorithmic coercibility)**.** $\Gamma \vdash \tau_1 \Longleftrightarrow \tau_2$ *holds iff* $\Gamma \vdash \tau_1 \sim \tau_2$ *holds.*

Proof. The direct sense is proved using Lemma 2.4.17. The converse is proved by induction on the equivalence judgment, using Lemma 2.4.16 for the case of reflexivity and Lemma 2.4.15 for the case of transitivity. □

We can finally proceed with the inversion lemmas on coercibility.

Lemma 2.4.19 (Inversion of coercibility)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'$, *then* $\Gamma \vdash \tau_1 \sim \tau_1'$ *and* $\Gamma \vdash \tau_2 \sim \tau_2'$.
- *If* $\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \sim (\ell_j' : \tau_j')^{j \in 1..m}$, *then* $n = m$ *and for every* $i \in 1..n$, $\ell_i = \ell_i'$ *and* $\Gamma \vdash \tau_i \sim \tau_i'$;
- *If* $\Gamma \vdash \forall \alpha.\tau \sim \forall \alpha.\tau'$, *then* $\Gamma, \forall \alpha \vdash \tau \sim \tau'$;
- *If* $\Gamma \vdash \exists \alpha.\tau \sim \exists \alpha.\tau'$, *then* $\Gamma, \forall \alpha \vdash \tau \sim \tau'$.

Proof. Using Lemma 2.4.18 and inversion of the normalization judgment. □

We now prove that coercibility is consistent, *i.e.* cannot equate distinct type constructors.

Lemma 2.4.20 (Consistency of coerciblity)**.** *The following assertions do* not *hold:*

- $\Gamma \vdash \tau_1 \to \tau_2 \sim \{(\ell_i : \tau'_i)^{i \in 1..n}\}$;
- $\Gamma \vdash \tau_1 \to \tau_2 \sim \forall \alpha.\tau$;
- $\Gamma \vdash \tau_1 \to \tau_2 \sim \exists \alpha.\tau$;
- $\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \sim \forall \alpha.\tau$;
- $\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \sim \exists \alpha.\tau$;
- $\Gamma \vdash \forall \alpha.\tau \sim \exists \beta.\tau'$.

Proof. Using Lemma 2.4.18 on the facing page and inversion of the normalization judgment. □

2.4.4 Properties of results and ϵ-reductions

In this section, we introduce ϵ-reductions, that characterize equivalent results, and permit a more precise statement of the progress lemma, that takes stuttering into account.

Definition 2.4.2 (ϵ-reductions)**.** We write $M \xrightarrow{\epsilon} M'$ when the reduction uses *only* Sigma-Sigma and Context. Conversely, we write $M \xrightarrow{\notin} M'$ when the reduction uses any rule *but* Sigma-Sigma. As usual, we use the notations $\xrightarrow{\epsilon}{}^{+}$ and $\xrightarrow{\notin}{}^{+}$ (respectively $\xrightarrow{\epsilon}{}^{\star}$ and $\xrightarrow{\notin}{}^{\star}$) for their transitive (respectively reflexive transitive) closures.

We now state that only ϵ-reduction can be applied to results, and that it transforms results into results.

Lemma 2.4.21. *The following assertions hold:*

- *If w is a result, then it* cannot *be $\notin$-reduced.*
- *If w is a result and $w \xrightarrow{\epsilon} M'$, then M' is also a result.*

The next lemma is necessary to prove our strengthened progress lemma: it states that the Σs in front of a well formed result can be reorganized at will using ϵ-reductions. In particular, on can put any selected one in front of all the other ones.

Lemma 2.4.22. *If $\Gamma \vdash w : \tau$ holds and $\exists \beta \in \Gamma$, then there exists α, τ' and w' such that $w \xrightarrow{\epsilon}{}^{\star} \Sigma\, \langle \beta \rangle\, (\alpha = \tau')\, w'$ and $\beta \notin \mathrm{ftv}(\tau')$.*

Proof. By induction on the typing judgment. Many cases are impossible because w is a result. It cannot be a value since it would contradict Lemma 2.3.4 on page 20. If w starts with a Σ, then either it is the right one (*i.e.* applied to β). Otherwise, we use induction hypothesis and then swap the two first Σs. □

2.4.5 Type soundness

We now proceed with the type soundness results. The proof of subject reduction proceeds by induction on the reduction relation. It heavily relies on the lemmas of Section 2.4.2 on page 24 and on Lemma 2.4.19 on the preceding page.

Proposition 2.4.23 (Subject reduction)**.** *If $\Gamma \vdash M : \tau$ and $M \longrightarrow M'$, then $\Gamma \vdash M' : \tau$.*

Progress states that every well-typed term is either a result (and then cannot $\notin$-reduce thanks to Lemma 2.4.21), or it *really* reduces, *i.e.* a $\notin$-reduction is possible, provided it is preceded by a finite, possibly empty, sequence of ϵ-reductions: this means that one can progress without stuttering using ϵ-reductions only. Progress is proved by induction on the typing derivation. It heavily relies on Lemma 2.4.22 and on Lemma 2.4.20 on the facing page.

Proposition 2.4.24 (Progress). *If* $\Gamma \vdash M : \tau$ *and* Γ *does not contain value variable bindings, then either* M *is a result, or there exists* M' *and* M'' *such that* $M \xrightarrow{\epsilon}{}^{\star} M' \xrightarrow{\notin} M''$.

The side condition that Γ does not contain any value variable is as usual. However, we cannot require the more restrictive hypothesis that Γ be empty, since evaluation takes place under the binders ν and $\exists$. Moreover, this allows to consider the reduction of *open* programs, *i.e.* programs with free type variables. This is the case of programs with abstract types, which come from unrestricted openings or open witness definitions. This closely corresponds to ML programs composed of modules with abstract types.

2.4.6 A mechanized proof of soundness

We developed a mechanized proof of soundness[1] (Proposition 2.4.23 on the previous page and Proposition 2.4.24) of a variant of $F^{\curlyvee}$ with The Coq proof assistant [Coq]. The difference with the version we described in this section is twofold:

- we considered *pairs* instead of records;
- there is only one zipping operator, that is the definition of the asymmetric zipping (Figure 2.4 on page 16), from which the fifth rule, that zips an existential on the left-hand side with nothing on the right-hand side is *removed*. In other words, its definition boils down to the following:

$$
\begin{array}{rcll}
\varepsilon \curlyvee \varepsilon & = & \varepsilon & \\
\Gamma_1, x : \tau \curlyvee \Gamma_2, x : \tau & = & (\Gamma_1 \curlyvee \Gamma_2), x : \tau & \text{if } x \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall \alpha \curlyvee \Gamma_2, \forall \alpha & = & (\Gamma_1 \curlyvee \Gamma_2), \forall \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall (\alpha = \tau) \curlyvee \Gamma_2, \forall (\alpha = \tau) & = & (\Gamma_1 \curlyvee \Gamma_2), \forall (\alpha = \tau) & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1 \curlyvee \Gamma_2, \exists \alpha & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha \curlyvee \Gamma_2, \forall \alpha & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2
\end{array}
$$

We made these changes for the sake of simplicity. It should not be hard to adapt the proof scripts to the definitions presented in the current manuscript. This slight change on zipping renders the system less symmetric, and consequently a bit less regular, since each typing rule with multiple premises uses the asymmetric version of zipping, whereas only the rule Let used the asymmetric version in the original definition. As a consequence, abstract type variables are automatically distributed to the right: they are available for use without the need use let-bindings. We think that the two systems are essentially equivalent, up to the introduction of intermediate let-bindings.

We chose to use the locally-nameless technique to represent terms with binders, and the cofinite quantification technique to express properties on such terms [ACP+08], with the great help of the Ott [SNO+10] and LNgen [AW] tools. Table 2.1 on the facing page reads the statistics of the Coq development. We first developed a soundness proof for System F in about one week, and then it took about one month to extend it to Core $F^{\curlyvee}$. The current development needs about 45 minutes to compile[2]: compilation time could surely be reduced, but the proof scripts were not produced with speed in mind. By contrast, the typechecking of the compiled Coq proof terms only needs less than 3 minutes.

While the above tools helped us a lot in most cases, it is worth noting that the current support for binders that they provided was too limited for our application: cofinite quantification does not, as implied, render renaming lemmas unnecessary, but also the support for extrusion or swapping of binders was completely absent, which forced us to prove low level lemmas, *i.e.* lemmas that involve

[1] The source files can be downloaded at the following URL: http://gallium.inria.fr/~montagu/proofs/FzipCore/
[2] On a machine running Linux 2.6.32, equipped with an Intel® Core™2 at 2.4 GHz CPU, and 4 Gb of RAM.

Specifications	Proofs	
3,000	1,450	Automatically generated
2,600	8,600	Manually produced
5,600	10,050	Total

Table 2.1: Statistics of the Coq soundness proof for $\mathsf{F}^{\forall}$ (lines of code).

$$
\begin{array}{rcl}
\lfloor x \rfloor & \triangleq & x \\
\lfloor \lambda(x:\tau)\, M \rfloor & \triangleq & \lambda(x)\, M \\
\lfloor M\ M' \rfloor & \triangleq & \lfloor M \rfloor\ \lfloor M' \rfloor \\
\lfloor \Lambda\alpha.\, M \rfloor & \triangleq & \lambda(x)\, \lfloor M \rfloor \quad \text{if } x \notin \mathrm{fv}(M) \\
\lfloor M\,\tau \rfloor & \triangleq & \lfloor M \rfloor\ (\lambda(x)\, x) \\
\lfloor \{(\ell_i = M_i)^{i\in 1..n}\} \rfloor & \triangleq & \{(\ell_i = \lfloor M_i \rfloor)^{i\in 1..n}\} \\
\left.\begin{array}{r}\lfloor \mathsf{let}\ x = M\ \mathsf{in}\ M' \rfloor \\ \lfloor \mathsf{unpack}\ M\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ M' \rfloor\end{array}\right\} & \triangleq & \mathsf{let}\ x = \lfloor M \rfloor\ \mathsf{in}\ \lfloor M' \rfloor
\end{array}
\qquad
\begin{array}{rcl}
\lfloor M.\ell \rfloor & \triangleq & \lfloor M \rfloor.\ell \\
\left.\begin{array}{r}\lfloor \nu\alpha.\, M \rfloor \\ \lfloor \exists\alpha.\, M \rfloor \\ \lfloor \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ M \rfloor \\ \lfloor \mathsf{open}\ \langle\alpha\rangle\ M \rfloor \\ \lfloor (M:\tau) \rfloor \\ \mathsf{pack}\ \langle \tau, M\rangle\ \mathsf{as}\ \exists\alpha.\tau' \end{array}\right\} & \triangleq & \lfloor M \rfloor
\end{array}
$$

Figure 2.14: Type erasure.

De Bruijn indices, that is, on the internal representation of binders. We think that, as the implementation of atoms in the UPenn library is hidden to the user, so should be the representation of binders. Efforts should be put in this direction. Besides, the cofinite quantification style drastically blurred the judgments that involved extrusion or swapping of binders, so that we were only convinced that our encoding was correct, once we proved that our judgments entailed some others, that were expressed in a more natural way.

A more detailed report of the encountered issues can be found in [Mon10].

2.4.7 Type erasure semantics

Type erasure is defined in Figure 2.14. Remark that the erasure of a generalization is a λ-abstraction, because in the semantics we gave for $\mathsf{F}^{\forall}$ in Figure 2.12 on page 22, no reduction happens under generalizations.

Just as for System F, $\mathsf{F}^{\forall}$ also enjoys a type erasure semantics, which is shown by the following simulation result:

Proposition 2.4.25 (Simulation with the untyped λ-calculus)**.** *The following assertions hold:*

- *If* $M \longrightarrow M'$, *then* $\lfloor M \rfloor = \lfloor M' \rfloor$ *or* $\lfloor M \rfloor \longrightarrow \lfloor M' \rfloor$;
- *If* $\lfloor M \rfloor \longrightarrow N$ *and* $\Gamma \vdash M : \tau$, *then there exists* N' *such that* $M \longrightarrow^{+} N'$ *and* $\lfloor N' \rfloor = N$.

The erasure of a term is defined in Figure 2.14. It is a standard erasure function, that blocks evaluation under Λs.

2.5 Adequacy with System F

In this section, we show the strong connection that exists between System F and Core $\mathsf{F}^{\forall}$: the latter is a conservative extension of the former, and allows more compositional programs to be written. We exhibit two encodings from and into System F, and show that they preserve types *and* meanings

$$
\begin{array}{rcll}
[\![x]\!]_F & \triangleq & x & \\
[\![\lambda(x:\tau)\,M]\!]_F & \triangleq & \lambda(x:\tau)\,[\![M]\!]_F & \\
[\![M\ M']\!]_F & \triangleq & [\![M]\!]_F\ [\![M']\!]_F & \\
[\![\mathsf{let}\ x = M\ \mathsf{in}\ M']\!]_F & \triangleq & \mathsf{let}\ x = [\![M]\!]_F\ \mathsf{in}\ [\![M']\!]_F & \\
[\![\Lambda\alpha.\,M]\!]_F & \triangleq & \Lambda\alpha.\,[\![M]\!]_F & \\
[\![M\,\tau]\!]_F & \triangleq & [\![M]\!]_F\,\tau & \\
[\![\{(\ell_i = M_i)^{i\in 1..n}\}]\!]_F & \triangleq & \{(\ell_i = [\![M_i]\!]_F)^{i\in 1..n}\} & \\
[\![M.\ell]\!]_F & \triangleq & [\![M]\!]_F.\ell & \\
[\![\mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M']\!]_F & \triangleq & \nu\alpha.\,\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ [\![M]\!]_F\ \mathsf{in}\ [\![M']\!]_F & \text{if } \alpha \notin \mathrm{ftv}(M) \\
[\![\mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as}\ \exists\alpha.\tau']\!]_F & \triangleq & \exists\alpha.\,\Sigma\,\langle\alpha\rangle\,(\alpha = \tau)\,([\![M]\!]_F : \tau') & \text{if } \alpha \notin \mathrm{ftv}(\tau) \cup \mathrm{ftv}(M)
\end{array}
$$

Figure 2.15: Encoding from F to $F^\forall$.

$$
\begin{array}{rcll}
[\![x]\!]^\Gamma_{F^\forall} & \triangleq & (\Gamma(x), x) & \\
[\![\lambda(x:\tau)\,M]\!]^\Gamma_{F^\forall} & \triangleq & (\tau \to \tau', \lambda(x:\tau)\,M') & \text{if } [\![M]\!]^{\Gamma, x:\tau}_{F^\forall} = (\tau', M') \\
[\![M_1\ M_2]\!]^\Gamma_{F^\forall} & \triangleq & (\tau_1', M_1'\ M_2') & \text{if } [\![M_1]\!]^\Gamma_{F^\forall} = (\tau_2' \to \tau_1', M_1') \text{ and } [\![M_2]\!]^\Gamma_{F^\forall} = (\tau_2', M_2') \\
[\![\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2]\!]^\Gamma_{F^\forall} & \triangleq & (\tau_2', \mathsf{let}\ x = M_1'\ \mathsf{in}\ M_2') & \text{if } [\![M_1]\!]^\Gamma_{F^\forall} = (\tau_1', M_1') \text{ and } [\![M_2]\!]^{\Gamma, x:\tau_1'}_{F^\forall} = (\tau_2', M_2') \\
[\![\Lambda\alpha.\,M]\!]^\Gamma_{F^\forall} & \triangleq & (\forall\alpha.\tau', \Lambda\alpha.\,M') & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\tau', M') \\
[\![M\,\tau]\!]^\Gamma_{F^\forall} & \triangleq & (\tau'[\alpha \leftarrow \tau], M'\,\tau) & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\forall\alpha.\tau', M') \\
[\![\{(\ell_i = M_i)^{i\in 1..n}\}]\!]^\Gamma_{F^\forall} & \triangleq & (\{(\ell_i : \tau_i')^{i\in 1..n}\}, \{(\ell_i = M_i')^{i\in 1..n}\}) & \text{if } [\![M_i]\!]^\Gamma_{F^\forall} = (\tau_i', M_i') \\
[\![M.\ell_k]\!]^\Gamma_{F^\forall} & \triangleq & (\tau_k', M'.\ell_k) & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\{(\ell_i : \tau_i')^{i\in 1..n}\}, M') \text{ and } k \in 1..n \\
[\![\nu\alpha.\,M]\!]^\Gamma_{F^\forall} & \triangleq & (\tau', \nu\alpha.\,M') & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\tau', M') \\
[\![\mathsf{open}\ \langle\alpha\rangle\ M]\!]^\Gamma_{F^\forall} & \triangleq & (\tau', \mathsf{open}\ \langle\alpha\rangle\ M') & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\exists\alpha.\tau', M') \\
[\![(M:\tau)]\!]^\Gamma_{F^\forall} & \triangleq & (\tau, (M':\tau)) & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\tau', M') \\
[\![\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,M]\!]^\Gamma_{F^\forall} & \triangleq & (\tau'[\alpha \leftarrow \beta], \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(M'[\alpha \leftarrow \tau] : \tau')) & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\tau', M') \\
[\![\exists\alpha.\,M]\!]^\Gamma_{F^\forall} & \triangleq & (\exists\alpha.\tau', \nu\alpha.\,\mathsf{let}\ x = M'\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, x\rangle\ \mathsf{as}\ \exists\alpha.\tau') & \text{if } [\![M]\!]^\Gamma_{F^\forall} = (\tau', M')
\end{array}
$$

Figure 2.16: Encoding from $F^\forall$ to F, stage 1: recovering packs.

of programs. These translations witness the static and dynamic correspondence between the two languages and highlight the gain of modularity that is brought by Core $F^\forall$.

2.5.1 From F to $F^\forall$

As mentioned in Section 2.2.1 on page 8, the encoding of pack and unpack is unsurprisingly straightforward. It preserves typing and abstraction as well as semantics: the encoding keeps the underlying untyped skeleton unchanged.

Lemma 2.5.1 (Translation preserves types)**.** *If* $\Gamma \vdash_F M : \tau$*, then* $\Gamma \vdash [\![M]\!]_F : \tau$.

Lemma 2.5.2 (Translation preserves semantics)**.** $\lfloor [\![M]\!]_F \rfloor = \lfloor M \rfloor$.

2.5.2 From $F^\forall$ to F

Conversely, it is also possible to globally reorganize every closed term of $F^\forall$ so that it uses (the encodings of) pack and unpack. We sketch out this transformation that consists in three stages:

$$
\begin{array}{rcl}
\mathcal{C}^\alpha & ::= & \mathsf{open}\,\langle\alpha\rangle\,M \;\mid\; \Sigma\,\langle\alpha\rangle\,(\beta=\tau)\,(M:\tau') \;\mid\; \mathcal{C}^\alpha\,M \;\mid\; M\,\mathcal{C}^\alpha \;\mid\; \mathcal{C}^\alpha\,\tau \;\mid\; \nu\beta.\,\mathcal{C}^\alpha \\
& \mid & \mathcal{C}^\alpha.\ell \;\mid\; \mathsf{open}\,\langle\beta\rangle\,\mathcal{C}^\alpha \;\mid\; \Sigma\,\langle\beta\rangle\,(\gamma=\tau)\,(\mathcal{C}^\alpha:\tau') \;\mid\; (\mathcal{C}^\alpha:\tau) \\
& \mid & \{(\ell_i = M_i)^{i\in I}\ ;\ \ell=\mathcal{C}^\alpha\ ;\ (\ell_j=M_j)^{j\in J}\} \;\mid\; \mathsf{let}\,x=M\,\mathsf{in}\,\mathcal{C}^\alpha \;\mid\; \mathsf{let}\,x=\mathcal{C}^\alpha\,\mathsf{in}\,M \\
& \mid & \mathsf{pack}\,\langle\tau,\mathcal{C}^\alpha\rangle\,\mathsf{as}\,\exists\beta.\tau' \;\mid\; \mathsf{unpack}\,\mathcal{C}^\alpha\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,M \;\mid\; \mathsf{unpack}\,M\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,\mathcal{C}^\alpha \\
& & \text{where } \beta,\gamma\neq\alpha
\end{array}
$$

$$
\begin{array}{rcll}
\nu\alpha.\,\mathsf{open}\,\langle\alpha\rangle\,M & \twoheadrightarrow & \mathsf{unpack}\,M\,\mathsf{as}\,\langle\alpha,x\rangle\,\mathsf{in}\,x & \\
\nu\alpha.\,\Sigma\,\langle\alpha\rangle\,(\beta=\tau)\,(M:\tau') & \twoheadrightarrow & \mathsf{unpack}\,\mathsf{pack}\,\langle\tau,M\rangle\,\mathsf{as}\ \exists\beta.\tau'\,\mathsf{as}\,\langle\alpha,x\rangle\,\mathsf{in}\,x & \text{if } \alpha\notin \mathrm{ftv}(\tau)\cup\mathrm{ftv}(\tau')\cup\mathrm{ftv}(M) \text{ and } \beta\notin\mathrm{ftv}(M) \\
\nu\alpha.\,(\mathcal{C}^\alpha\,M) & \twoheadrightarrow & (\nu\alpha.\,\mathcal{C}^\alpha)\,M & \text{if } \alpha\notin\mathrm{ftv}(M) \\
\nu\alpha.\,(M\,\mathcal{C}^\alpha) & \twoheadrightarrow & M\,(\nu\alpha.\,\mathcal{C}^\alpha) & \text{if } \alpha\notin\mathrm{ftv}(M) \\
\nu\alpha.\,(\mathcal{C}^\alpha\,\tau) & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\,x=\mathcal{C}^\alpha\,\mathsf{in}\,x\,\tau & \\
\nu\alpha.\,\nu\beta.\,M & \twoheadrightarrow & \nu\beta.\,\nu\alpha.\,M & \text{if } \alpha\neq\beta \\
\nu\alpha.\,\mathcal{C}^\alpha.\ell & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\,x=\mathcal{C}^\alpha\,\mathsf{in}\,x.\ell & \text{if } \alpha\neq\beta \\
\nu\alpha.\,\mathsf{open}\,\langle\beta\rangle\,\mathcal{C}^\alpha & \twoheadrightarrow & \mathsf{open}\,\langle\beta\rangle\,\nu\alpha.\,\mathcal{C}^\alpha & \text{if } \alpha\neq\beta \\
\nu\alpha.\,\Sigma\,\langle\beta\rangle\,(\gamma=\tau)\,(\mathcal{C}^\alpha:\tau') & \twoheadrightarrow & \Sigma\,\langle\beta\rangle\,(\gamma=\tau)\,(\nu\alpha.\,\mathcal{C}^\alpha:\tau') & \text{if } \alpha\notin\{\beta,\gamma\}\cup\mathrm{ftv}(\tau)\cup\mathrm{ftv}(\tau') \\
\nu\alpha.\,(\mathcal{C}^\alpha:\tau) & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\,x=\mathcal{C}^\alpha\,\mathsf{in}\,(x:\tau) & \\
\nu\alpha.\,\{(\ell_i=M_i)^{i\in1..n}\ ;\ \ell=\mathcal{C}^\alpha\ ;\ (\ell'_j=M'_j)^{j\in1..m}\} & \twoheadrightarrow & \{(\ell_i=M_i)^{i\in1..n}\ ;\ \ell=\nu\alpha.\,\mathcal{C}^\alpha\ ;\ (\ell'_j=M'_j)^{j\in1..m}\} & \text{if } \alpha\notin\mathrm{ftv}(M_i)^{i\in1..n}\cup\mathrm{ftv}(M'_j)^{j\in1..m} \\
\nu\alpha.\,(\mathsf{let}\,x=M\,\mathsf{in}\,\mathcal{C}^\alpha) & \twoheadrightarrow & \mathsf{let}\,x=M\,\mathsf{in}\,\nu\alpha.\,\mathcal{C}^\alpha & \text{if } \alpha\notin\mathrm{ftv}(M) \\
\nu\alpha.\,\mathsf{pack}\,\langle\tau,\mathcal{C}^\alpha\rangle\,\mathsf{as}\,\exists\beta.\tau' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\,x=\mathcal{C}^\alpha\,\mathsf{in}\,\mathsf{pack}\,\langle\tau,x\rangle\,\mathsf{as}\,\exists\beta.\tau' & \\
\nu\alpha.\,\mathsf{unpack}\,\mathcal{C}^\alpha\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\,y=\mathcal{C}^\alpha\,\mathsf{in}\,\mathsf{unpack}\,y\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,M & \text{if } y \text{ fresh} \\
\nu\alpha.\,\mathsf{unpack}\,M\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,\mathcal{C}^\alpha & \twoheadrightarrow & \mathsf{unpack}\,M\,\mathsf{as}\,\langle\beta,x\rangle\,\mathsf{in}\,\nu\alpha.\,\mathcal{C}^\alpha & \text{if } \alpha\notin\{\beta\}\cup\mathrm{ftv}(M)
\end{array}
$$

Figure continues on page ??.

Figure 2.17: Encoding from $\mathsf{F}^{\forall}$ to F, stage 2: recovering unpack s (congruence rules are omitted).

$$
\begin{array}{rcll}
\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ M\ \mathsf{in}\ M' & \rightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = & \rightarrow & \mathsf{unpack}\ \mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as} & \text{if } \alpha \notin \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\tau') \cup \mathrm{ftv}(M) \\
\quad \Sigma\ \langle\alpha\rangle\ (\beta = \tau)\ (M : \tau')\ \mathsf{in}\ M' & & \quad \exists\beta.\tau'\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = \mathcal{C}^\alpha\ M\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ M\ \mathsf{in}\ M' & \text{if } y \text{ is fresh} \\
\nu\alpha.\,\mathsf{let}\ x = M\ \mathcal{C}^\alpha\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = M\ y\ \mathsf{in}\ M' & \text{if } y \text{ is fresh} \\
\nu\alpha.\,\mathsf{let}\ x = \mathcal{C}^\alpha\ \tau\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ \tau\ \mathsf{in}\ M' & \text{if } y \text{ is fresh} \\
\nu\alpha.\,\mathsf{let}\ x = \nu\beta.\,\mathcal{C}^\alpha\ \mathsf{in}\ M' & \rightarrow & \nu\beta.\,\nu\alpha.\,\mathsf{let}\ x = \mathcal{C}^\alpha\ \mathsf{in}\ M' & \text{if } \beta \notin \{\alpha\} \cup \mathrm{ftv}(M') \\
\nu\alpha.\,\mathsf{let}\ x = \mathcal{C}^\alpha.\ell\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y.\ell\ \mathsf{in}\ M' & \text{if } y \text{ is fresh} \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}\ \langle\beta\rangle\ \mathcal{C}^\alpha\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in} & \text{if } y \text{ is fresh}, \alpha \neq \beta \\
 & & \quad \mathsf{let}\ x = \mathsf{open}\ \langle\beta\rangle\ y\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in} & \text{if } y \text{ is fresh}, \beta \notin \{\alpha\} \cup \mathrm{ftv}(\mathcal{C}^\alpha) \\
\quad \Sigma\ \langle\gamma\rangle\ (\beta = \tau)\ (\mathcal{C}^\alpha : \tau')\ \mathsf{in}\ M' & & \quad \mathsf{let}\ x = \Sigma\ \langle\gamma\rangle\ (\beta = \tau)\ (y : \tau')\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = (\mathcal{C}^\alpha : \tau)\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = (y : \tau)\ \mathsf{in}\ M' & \text{if } y \text{ is fresh} \\
\nu\alpha.\,\mathsf{let}\ x = \{(\ell_i = M_i)^{i\in 1..n}\ ; & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = & \text{if } (y_i)^{i\in 1..n}, y \text{ are fresh} \\
\quad \ell = \mathcal{C}^\alpha\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' & & \quad \{(\ell_i = M_i)^{i\in 1..n}\ ;\ \ell = y\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{let}\ y = M\ \mathsf{in}\ \mathcal{C}^\alpha\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = M\ \mathsf{in}\ \mathsf{let}\ x = \mathcal{C}^\alpha\ \mathsf{in}\ M' & \text{if } y \notin \mathrm{fv}(M') \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ M\ \mathsf{in}\ M' & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = M\ \mathsf{in}\ M' & \text{if } y \notin \mathrm{fv}(M') \\
\nu\alpha.\,\mathsf{let}\ x = & \rightarrow & \nu\alpha.\,\mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in} & \text{if } y \text{ is fresh} \\
\quad \mathsf{pack}\ \langle\tau, \mathcal{C}^\alpha\rangle\ \mathsf{as}\ \exists\beta.\tau'\ \mathsf{in}\ M' & & \quad \mathsf{let}\ x = \mathsf{pack}\ \langle\tau, y\rangle\ \mathsf{as}\ \exists\beta.\tau'\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = & \rightarrow & \nu\alpha.\,\mathsf{unpack}\ \mathcal{C}^\alpha\ \mathsf{as}\ \langle\beta, y\rangle\ \mathsf{in} & \text{if } y \notin \mathrm{fv}(M'), \beta \notin \mathrm{ftv}(M') \\
\quad \mathsf{unpack}\ \mathcal{C}^\alpha\ \mathsf{as}\ \langle\beta, y\rangle\ \mathsf{in}\ M\ \mathsf{in}\ M' & & \quad \mathsf{let}\ x = M\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = & \rightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \langle\beta, y\rangle\ \mathsf{in} & \text{if } y \notin \mathrm{fv}(M'), \beta \notin \mathrm{ftv}(M') \\
\quad \mathsf{unpack}\ M\ \mathsf{as}\ \langle\beta, y\rangle\ \mathsf{in}\ \mathcal{C}^\alpha\ \mathsf{in}\ M' & & \quad \nu\alpha.\,\mathsf{let}\ x = \mathcal{C}^\alpha\ \mathsf{in}\ M' &
\end{array}
$$

Figure 2.17: Encoding from $\mathsf{F}^\forall$ to F, stage 2 (continued).

Prelude First, we apply the transformation $[\![\cdot]\!]^{\Gamma}_{\mathsf{F}^\forall}$, defined in Figure 2.16 on page 30, that recovers the pack constructs from the existential closure constructs;

Main part Then, we extrude opens and Σs using let-bindings and intrude νs to recover the unpack constructs, by completely applying for every ν the rewriting rules defined in Figure 2.17 on the previous page;

Postlude Finally, we remove all coercions, since they become unnecessary.

All stages but the second one are compositional. We now review the different steps in more details. Notice that we use an extended syntax of terms: to the syntax of $\mathsf{F}^\forall$ terms, we add the constructs pack and unpack of System F.

First stage: prelude

The first stage of the translation is defined in Figure 2.16 on page 30. It can be seen as a normalization pass: it computes the current type, inserts coercions at the current type under Σs and unfolds witness definitions within their scopes, and finally replaces uses of the $\exists$-construct with uses of pack. The main ingredients of this stage are visible in the last two lines of Figure 2.16.

Remark that the environment in the translation only contain value variable bindings, since other sorts of bindings are not needed.

It could also have been possible to define this stage as a type-directed translation, but this definition does not require the terms to be welltyped.

This transformation enjoys the following properties:

Lemma 2.5.3 (Weakening). *If* $[\![M]\!]^{\Gamma}_{F^Y} = (\tau, M')$*, then for every* Γ' *such that* $\forall x \in \operatorname{dom}\Gamma, \Gamma'(x) = \Gamma(x)$*, we have* $[\![M]\!]^{\Gamma'}_{F^Y} = (\tau, M')$.

Lemma 2.5.4 (Productivity). *If* $\Gamma \vdash M : \tau$*, then there exists* τ' *and* M' *such that* $[\![M]\!]^{\Gamma}_{F^Y} = (\tau', M')$.

Lemma 2.5.5 (Preservation of types). *If* $\Gamma \vdash M : \tau$ *and* $[\![M]\!]^{\Gamma}_{F^Y} = (\tau', M')$*, then* $\tau = \tau'$ *and* $\Gamma \vdash M' : \tau'$.

Definition 2.5.1 (let-reduction of erased terms). let-reduction of erased terms, denoted by $\xrightarrow{\mathsf{let}}$, is the closure of the rule $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \longrightarrow M_2[x \leftarrow M_1]$ under any context.

Lemma 2.5.6 (Preservation of semantics). *If* $[\![M]\!]^{\Gamma}_{F^Y} = (\tau', M')$*, then* $\lfloor M' \rfloor \xrightarrow{\mathsf{let}}{}^{\star} \lfloor M \rfloor$.

Second stage: main transformation

The second stage of the translation is defined in Figure 2.17 on page 31. It is defined in a small-step manner, as a set of rewrite rules. Each rewriting step tends to move closer together a ν and its corresponding open or Σ, *i.e.* the site of introduction of an existential type variable and the site of its use. This will eventually lead to the replacement of those constructs with unpack.

Definition 2.5.2. We say that a rewrite rule is *rooted* at α if its left-hand side is of the form $\nu\alpha.\, \mathcal{C}^{\alpha}$.

The rewrite rules are intended to be used to eliminate each ν one after the other. For each of them, the rewrite rules intrude the ν when possible, or introduce an extra let otherwise, and eventually insert an unpack. The rewrite rules rely on the definition of $\mathcal{C}^{\alpha}$, which is a grammar of contexts, that locate the use of the variable α. It is worth noting that if $\Gamma \vdash M : \tau$ and $\exists\alpha \in \Gamma$, then there exists a *unique* $\mathcal{C}^{\alpha}$ such that $M = \mathcal{C}^{\alpha}$: this is due to the linear condition on the use of $\exists\alpha$, that is enforced by the typing judgment.

For a given α, the rewriting rules rooted at α always terminate, since they decrease the distance between the introduction and the elimination of α. More formally, the *depth* of the type variable α decreases.

Definition 2.5.3 (Depth). The *depth* of the existential type variable α in the context $\mathcal{C}^{\alpha}$, denoted by $\operatorname{depth}^{\alpha} \mathcal{C}^{\alpha}$, is defined as follows:

$$\left.\begin{array}{r} \operatorname{depth}^{\alpha} \mathsf{open}\ \langle\alpha\rangle\ M \\ \operatorname{depth}^{\alpha} \Sigma\ \langle\alpha\rangle\ (\beta = \tau)\ M \end{array}\right\} \triangleq 0$$

$$\operatorname{depth}^{\alpha} \mathsf{let}\ x = \mathcal{C}^{\alpha}\ \mathsf{in}\ M \triangleq 1 + \operatorname{depth}^{\alpha} \mathcal{C}^{\alpha}$$

$$\left.\begin{array}{r} \operatorname{depth}^{\alpha} \mathcal{C}^{\alpha}\,\tau \\ \operatorname{depth}^{\alpha} \nu\beta.\, \mathcal{C}^{\alpha} \\ \operatorname{depth}^{\alpha} \mathcal{C}^{\alpha}.\ell \\ \operatorname{depth}^{\alpha} \mathsf{open}\ \langle\beta\rangle\ \mathcal{C}^{\alpha} \\ \operatorname{depth}^{\alpha} \Sigma\ \langle\beta\rangle\ (\gamma = \tau)\ M \\ \operatorname{depth}^{\alpha} (\mathcal{C}^{\alpha} : \tau) \end{array}\right\} \triangleq \left\{\begin{array}{c} \operatorname{depth}^{\alpha} \{(\ell_i = M_i)^{i \in 1..n}\ ; \\ \quad \ell = \mathcal{C}^{\alpha}\ ;\ (\ell'_j = M'_j)^{j \in 1..n}\} \\ \operatorname{depth}^{\alpha} \mathsf{let}\ x = M\ \mathsf{in}\ \mathcal{C}^{\alpha} \\ \operatorname{depth}^{\alpha} \mathsf{pack}\ \langle\tau, \mathcal{C}^{\alpha}\rangle\ \mathsf{as}\ \exists\beta.\tau' \\ \operatorname{depth}^{\alpha} \mathsf{unpack}\ M\ \mathsf{as}\ \langle\beta, x\rangle\ \mathsf{in}\ \mathcal{C}^{\alpha} \\ \operatorname{depth}^{\alpha} \mathsf{unpack}\ \mathcal{C}^{\alpha}\ \mathsf{as}\ \langle\beta, x\rangle\ \mathsf{in}\ M \end{array}\right\} \triangleq 2 + \operatorname{depth}^{\alpha} \mathcal{C}^{\alpha} \quad \text{when } \alpha \neq \beta$$

Notation. In the following, we write $M[N]$ if the term N occurs as a subterm of M.

For a given root α, the rewriting relation terminates:

Lemma 2.5.7 (Termination). *If* $\nu\alpha.\, \mathcal{C}_1^{\alpha} \twoheadrightarrow M[\nu\alpha.\, \mathcal{C}_2^{\alpha}]$ *using a rule rooted at* α*, then the depth of* α *strictly decreases, that is* $\operatorname{depth}^{\alpha} \mathcal{C}_2^{\alpha} < \operatorname{depth}^{\alpha} \mathcal{C}_1^{\alpha}$ *holds. Hence the rewrite rules rooted at a given type variable terminate.*

The next lemma is useful to complete the transformation: if applied to an innermost root α, then the number of νs strictly decreases, once the rules rooted at α have all been applied.

Lemma 2.5.8. *If* $\nu\alpha.\, \mathcal{C}_1^\alpha \twoheadrightarrow M[\nu\alpha.\, \mathcal{C}_2^\alpha]$ *using a rule rooted at* α*, and if* $\mathcal{C}_1^\alpha$ *is* ν*-free, then so is* $\mathcal{C}_2^\alpha$*.*

The transformation is well defined on wellformed terms that are not ν-free.

Lemma 2.5.9 (Progress)**.** *If* $\Gamma \vdash N[\nu\alpha.\, M] : \tau$*, then* M *is of the form* $\mathcal{C}^\alpha$ *and there exists a rule rooted at* α *that rewrites* $\nu\alpha.\, M$*. Moreover, this rule is unique.*

The transformation is type preserving.

Lemma 2.5.10 (Preservation of types)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \twoheadrightarrow M'$*, then* $\Gamma \vdash M' : \tau$*.*

Then, the next two lemmas show that the transformation eventually eliminates the restrictions.

Lemma 2.5.11 (Productivity)**.** *If* $\Gamma \vdash N[\nu\alpha.\, M] : \tau$*, then there exists* M'*,* M_1*,* M_2*, and* x *such that:*

- $\nu\alpha.\, M \twoheadrightarrow^+ M'[\mathsf{unpack}\ M_1\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M_2]$ *using only rules that are rooted at* α*;*
- *if* M *is* ν*-free, then so are* M'*,* M_1 *and* M_2*.*

Proof. By induction on $\mathrm{depth}^\alpha M$, using Lemma 2.5.9 and Lemma 2.5.10. □

Lemma 2.5.12 (Correctness)**.** *If* $\Gamma \vdash M : \tau$*, then there exists* M' *such that* $M \twoheadrightarrow^\star M'$ *and* M' *is* ν*-free.*

Proof. By applying Lemma 2.5.11 following an innermost strategy, so that the number of νs strictly decreases. □

It has been already shown that the translation preserves types; it is also semantics-preserving.

Definition 2.5.4. $\xrightarrow{\mathsf{let}}$ is the smallest reduction relation on pure λ-terms that is closed under any context and so that $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \xrightarrow{\mathsf{let}} M_2[x \leftarrow M_1]$ holds.

Lemma 2.5.13 (Preservation of semantics)**.** *If* $M \twoheadrightarrow M'$*, then* $\lfloor M' \rfloor \xrightarrow{\mathsf{let}}{}^\star \lfloor M \rfloor$*.*

Notice that we did not take care of keeping the same evaluation order between a source term and its image, but this is achievable. For instance, consider the following rule, taken from Figure 2.17 on page 31:

$$\begin{array}{l} \nu\alpha.\, \mathsf{let}\ x = \{(\ell_i = M_i)^{i\in 1..n}\ ; \\ \quad \ell = \mathcal{C}^\alpha\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' \end{array} \twoheadrightarrow \begin{array}{l} \nu\alpha.\, \mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \\ \quad \{(\ell_i = M_i)^{i\in 1..n}\ ;\ \ell = y\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' \end{array}$$

On the left hand side, M will be evaluated *after* the subterms $M_1, \ldots, M_n$, whereas on the right hand side M will be evaluated *before* the subterms $M_1, \ldots, M_n$. But we could replace the previous rewriting rule with the following one:

$$\begin{array}{l} \nu\alpha.\, \mathsf{let}\ x = \{(\ell_i = M_i)^{i\in 1..n}\ ; \\ \quad \ell = \mathcal{C}^\alpha\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' \end{array} \twoheadrightarrow \begin{array}{l} \mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ x_n = M_n\ \mathsf{in} \\ \nu\alpha.\, \mathsf{let}\ y = \mathcal{C}^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \\ \quad \{(\ell_i = x_i)^{i\in 1..n}\ ;\ \ell = y\ ;\ (\ell'_j = M'_j)^{j\in 1..m}\}\ \mathsf{in}\ M' \\ \text{if } x_1, \ldots, x_n \text{ are fresh, } \alpha \notin \mathrm{ftv}(M_1) \cup \cdots \cup \mathrm{ftv}(M_n) \end{array}$$

This would restore the evaluation order, at the cost of a more verbose rule.

Third stage: postlude

The last stage of the translation is again a normalization step: it consists in removing the coercions. Let us write $(M)^\circ$ to denote the erasure of coercions from M. This is made possible thanks to the following lemmas.

Lemma 2.5.14. *If $\Gamma \vdash M : \tau$ holds and Γ is equation-free* (i.e. $\mathrm{dom}^\simeq\, \Gamma = \emptyset$) *and M is Σ-free, then every context occurring in the derivation of $\Gamma \vdash M : \tau$ is also equation-free.*

Lemma 2.5.15. *If $\Gamma \vdash \tau_1 \sim \tau_2$ and Γ is equation-free, then $\tau_1 = \tau_2$.*

Since we only translate well-typed terms that are closed with respect to type variables, they are also typable in an equation-free and pure environment. Moreover, after the second stage of transformation, terms are Σ-free, since they are ν-free and $\exists$-free and well-typed in a pure environment. As a consequence, the coercions that occur within those terms can only be instances of the identity, that is of the syntactic equality, and can consequently been eliminated.

The third stage enjoys the following properties.

Lemma 2.5.16 (Preservation of types)**.** *If $\Gamma \vdash M : \tau$ where $\Gamma\,\mathrm{pure}$ and Γ is equation-free and M is ν-free and $\exists$-free, then $\Gamma \vdash (M)^\circ : \tau$.*

Lemma 2.5.17 (Preservation of semantics)**.** $\lfloor (M)^\circ \rfloor = \lfloor M \rfloor$.

Properties of the whole transformation

Combining the properties of each stage, we can finally prove that terms of $\mathsf{F}^\forall$ can be translated to F:

Proposition 2.5.18 (Translation to System F)**.** *If $\Gamma \vdash M : \tau$ and $\Gamma\,\mathrm{pure}$ and Γ is equation-free, then there exists N such that $\Gamma \vdash_{\mathsf{F}} N : \tau$ and $\lfloor N \rfloor \xrightarrow{\mathsf{let}}{}^\star \lfloor M \rfloor$.*

Corollary 2.5.19. *Every closed well-typed term of $\mathsf{F}^\forall$ can be translated to a closed well-typed term of System F that has the same type and the same behavior.*

This result highlights the increase of modularity brought by $\mathsf{F}^\forall$ over System F: the translation reorganizes the term by introducing intermediate let-bindings, so that it fits in System F. In other words, $\mathsf{F}^\forall$ allows for organizing the code more freely than F does.

One could wonder whether the translation also preserves a certain notion of *abstraction*, or more informally: *are the witnesses kept hidden by the translation?* It is not obvious how to formally state the intended property, and we did not investigate on this question. One can however argue in favor of the preservation of abstraction:

- the inserted coercions (Figure 2.16 on page 30) are always there to keep the type unchanged: they restore the type that held before the substitution of the witness;
- the Σs, which delimit the scopes of type equations, are never extruded;
- the inserted let-bindings use *fresh variables*: as a consequence, the pieces of term that are extruded with a let-binding are only visible to the subterms they are extracted from.

A formal investigation of abstraction in $\mathsf{F}^\forall$ and of its translation for System F could certainly be done with the use of bisimulation techniques [Mit86, Mit91, SP04], of logical relations [CH07], or of colored brackets [Pes08, GMZ00, LPSW03].

2.5.3 The logical facet

By erasing the terms from the typing rules, we can consider the logic underlying core $\mathsf{F}^{\curlyvee}$: not only the expressible formulas are exactly those of second-order arithmetic, but also we can deduce from the translations above that the *valid formulas are identical*. In particular, $\mathsf{F}^{\curlyvee}$'s logic is consistent. Moreover, since the reduction steps are increased by the translation and since the untyped skeletons of System F terms are terminating, the untyped skeleton of every closed program of $\mathsf{F}^{\curlyvee}$ is also terminating. In addition, the fact that the untyped skeleton of the image let-reduces to the untyped skeleton of the source essentially tells us that the two pieces of program compute *the same things* and *in the same way*: the translation to System F just performs a reorganization of the type derivation.

Hence, the correspondence with System F is twofold: it holds on the static as well as on the dynamic viewpoint, which connects $\mathsf{F}^{\curlyvee}$ with System F in a very tight manner.

The gain of modularity brought by core $\mathsf{F}^{\curlyvee}$ in terms of programming can be read back in terms of proofs: it allows greater flexibility in assembling partial proofs (*i.e.* with abstract types), where environments are *zipped* when combining proof-terms.

One can wonder what is the logical status of the new typing rules that we introduced in $\mathsf{F}^{\curlyvee}$:

- rule Coerce has the form of an explicit subtyping rule with an empty computational content, which is corroborated by Section 2.2.3 on page 14;
- rule Exists is the right introduction rule for the existential quantifier, whereas rule Open is its elimination rule, which is corroborated by the fact that $\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\, w$ is a redex;
- rule Sigma is also an introduction rule, whereas rule Nu is its elimination rule, which is again corroborated by the fact that $\nu\beta.\, \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ w$ is a redex.

2.6 Extensions of $\mathsf{F}^{\curlyvee}$

In this section we consider several extensions for $\mathsf{F}^{\curlyvee}$, namely: the addition of a weakening rule, more liberal type equations, handling of the double vision problem, support of recursive types, and support of recursive values. For each of them, the soundness properties (Proposition 2.4.23 and Proposition 2.4.24 on page 28) of $\mathsf{F}^{\curlyvee}$ extend, and are sketched in Section 2.6.6 on page 43.

2.6.1 Weakening

The weakening property (Lemma 2.4.7 on page 24) is restricted to bindings that do not depend on existential items. While this is sufficient to prove subject reduction, is also has consequences on the expressiveness of $\mathsf{F}^{\curlyvee}$ itself.

The following example is not accepted, but this is not due to the lack of weakening: the asymmetric zipping is responsible, and a solution is given in Section 2.6.2 to allow more liberal type equations.

$$\begin{array}{l}\mathsf{let}\ f = \lambda(x : \beta)\, x\ \mathsf{in}\\ \quad \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\ (1 : \alpha)\end{array}$$

The existential variable is indeed not present in the context that types the function f, because zipping distributes a universal variable only to the right. However, β is required to be in the environment to type f, since it is a free type variable of f.

The next example is however a relevant instance of the lack of weakening:

$$\begin{array}{l}\nu\beta.\, \Sigma\ \langle\delta\rangle\ (\gamma = \beta)\\ \quad \mathsf{let}\ x = \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\ (1 : \alpha)\ \mathsf{in}\\ \quad (x : \gamma)\end{array}$$

The above piece of code is not accepted, because to type the definition of x, the Σ requires that β is not present in the environment to type $(1 : \alpha)$, whereas the equation $\forall(\gamma = \beta)$ is already in the environment. As a consequence, the above example is rejected. Another way to explain this behavior is that one must respect the order of definitions of witnesses: one tries to define δ and γ using β, that is not yet defined. The following piece of code, is, however, accepted: the only difference is that the Σ, that previously encompassed the whole term, has been intruded.

$$\nu\beta.\,\mathsf{let}\ x = \Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,(1:\alpha)\ \mathsf{in}\\ \Sigma\,\langle\delta\rangle\,(\gamma = \beta)\,(x:\gamma)$$

We think the former example was rejected for a bad reason, that is, because of a technical artifact, and that might seem difficult to apprehend for a non specialist of $\mathsf{F}^{\forall}$.

To resolve this awkwardness of the system, it suffices to add a weakening rule to the system, that allows to remove elements from the environment to type a given term, as long as the removed items are *pure*. The weakening judgment is defined in Figure 2.21 on page 44, and is used in the rule Weaken in Figure 2.20 on page 44. This way, one can remove the extra equation $\forall(\gamma = \beta)$ before typing $\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,(1:\alpha)$. Notice that it also permits to type the same piece of code, where x has been inlined.

The examples we gave proceed to a renaming of an abstract type, that is: they transform a derivation of $\Gamma, \exists\beta \vdash M : \tau$ into a derivation of $\Gamma, \exists\delta \vdash M' : \tau[\beta \leftarrow \delta]$ where δ is fresh. This technique is described in a more general form that we call relocation in Section 4.2.2 on page 119, and is heavily used, in the translation of Section 4.3.2 on page 122, in Chapter 4.

2.6.2 More liberal equations

Core $\mathsf{F}^{\forall}$ imposes a simple but inconveniently strong restriction to force type equations to be acyclic.

In this section we present a more general technique to control recursive types, by enriching the structure of typing environments in a natural way: we no longer consider them as sequences, *i.e. totally* ordered sets, but as *partially* ordered sets, where the order relation expresses dependencies between bindings and is required to be *acyclic*, which means that no binding can (transitively) depend on itself. Failure to satisfy this condition prevents the zipping of two environments (the zipped environment is undefined).

More specifically, a typing environment Γ is a dag represented as a pair $(\mathcal{E}, \prec)$ of a finite set of bindings $\mathcal{E}$ and an acyclic (or anti-reflexive) transitive relation $\prec$ on $\operatorname{dom}\mathcal{E}$, *i.e.* there exists no binding b such that $\operatorname{dom} b \prec \operatorname{dom} b$. We sometimes write $b \prec b'$ instead of $\operatorname{dom} b \prec \operatorname{dom} b'$. If $b \prec b'$, we say that b depends on b'.

Definition 2.6.1. We consider that two environments $(\mathcal{E}_1, \prec_1)$ and $(\mathcal{E}_2, \prec_2)$ are equal when the sets $\mathcal{E}_1$ and $\mathcal{E}_2$ are (extensionally) equal and the relations $\prec_1$ and $\prec_2$ are equivalent.

Note. We take care that all the definitions and properties involving environments are closed under the equality on environments.

We use the following notation for composing and decomposing typing environments so that typing rules look familiar:

Notation. We write $\Gamma_1, (b \prec \mathcal{D}), \Gamma_2$ to denote an environment that contains the bindings of $\Gamma_1 \uplus b \uplus \Gamma_2$, when no binding in Γ_1 depends on b, and b does not depend on bindings of Γ_2, and $\mathcal{D}$ is the set of bindings b depends on. In particular, when Γ_2 is empty, b is minimal for the dependency relation.

In the rest of the document, we will only consider cases, where dependencies hold with respect to type variables only: we do not make bindings depend on other value variable bindings. However, value variables can depend on type variables, as in rule Let.

$$
\begin{array}{rcll}
\emptyset \curlyvee \emptyset & \triangleq & \emptyset & \\
(\mathfrak{b} \cup \mathcal{E}_1) \curlyvee (\mathfrak{b} \cup \mathcal{E}_2) & \triangleq & \mathfrak{b} \cup (\mathcal{E}_1 \curlyvee \mathcal{E}_2) & \text{if } \operatorname{dom} \mathfrak{b} \notin \operatorname{dom} \mathcal{E}_1 \cup \operatorname{dom} \mathcal{E}_2 \\
 & & & \text{and } \mathfrak{b} \neq \exists \beta \\
(\{\exists \alpha\} \cup \mathcal{E}_1) \curlyvee (\{\forall \alpha\} \cup \mathcal{E}_2) & \triangleq & \{\exists \alpha\} \cup (\mathcal{E}_1 \curlyvee \mathcal{E}_2) & \text{if } \alpha \notin \operatorname{dom} \mathcal{E}_1 \cup \operatorname{dom} \mathcal{E}_2 \\
(\{\forall \alpha\} \cup \mathcal{E}_1) \curlyvee (\{\exists \alpha\} \cup \mathcal{E}_2) & \triangleq & \{\exists \alpha\} \cup (\mathcal{E}_1 \curlyvee \mathcal{E}_2) & \text{if } \alpha \notin \operatorname{dom} \mathcal{E}_1 \cup \operatorname{dom} \mathcal{E}_2 \\
(\mathfrak{b} \cup \mathcal{E}_1) \curlyvee \mathcal{E}_2 & \triangleq & \mathfrak{b} \cup (\mathcal{E}_1 \curlyvee \mathcal{E}_2) & \text{if } \operatorname{dom} \mathfrak{b} \notin \operatorname{dom} \mathcal{E}_1 \cup \operatorname{dom} \mathcal{E}_2 \\
\mathcal{E}_1 \curlyvee (\mathfrak{b} \cup \mathcal{E}_2) & \triangleq & \mathfrak{b} \cup (\mathcal{E}_1 \curlyvee \mathcal{E}_2) & \text{if } \operatorname{dom} \mathfrak{b} \notin \operatorname{dom} \mathcal{E}_1 \cup \operatorname{dom} \mathcal{E}_2
\end{array}
$$

Figure 2.18: Zipping of sets of bindings

Definition 2.6.2 (Zipping)**.** Let Γ_1 and Γ_2 be two typing environments of the form $(\mathcal{E}_1, \prec_1)$ and $(\mathcal{E}_2, \prec_2)$. Let $\prec$ be the transitive closure $(\prec_1 \cup \prec_2)^+$. If $\prec$ is acyclic, the zipping of Γ_1 and Γ_2, written $\Gamma_1 \curlyvee \Gamma_2$, is $(\mathcal{E}_1 \curlyvee \mathcal{E}_2, \prec)$, where $\mathcal{E}_1 \curlyvee \mathcal{E}_2$ is defined in Figure 2.18. The zipping of Γ_1 and Γ_2 is undefined if $\prec$ is not acyclic or if $\mathcal{E}_1 \curlyvee \mathcal{E}_2$ is undefined.

The last two items of Figure 2.18 perform an implicit weakening on each environment. This has the effect of refining the detection of cycles, and will be illustrated below.

Lemma 2.6.1. *The zipping operator enjoys the following properties:*

Associativity: $\Gamma_1 \curlyvee (\Gamma_2 \curlyvee \Gamma_3)$ *is defined iff* $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee \Gamma_3$ *is defined, and in this case, the two environments are equal;*

Commutativity: $\Gamma_1 \curlyvee \Gamma_2$ *is defined iff* $\Gamma_2 \curlyvee \Gamma_1$ *is defined, and in this case, the two environments are equal;*

Distributivity: *if* Γ_1 pure, *then* $\Gamma_1 \curlyvee (\Gamma_2 \curlyvee \Gamma_3)$ *is defined iff* $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee (\Gamma_1 \curlyvee \Gamma_3)$ *is defined, and in this case, the two environments are equal.*

Observe that the zipping properties are much more regular in this setting than they were in Core $\mathsf{F}^{\curlyvee}$ (see Lemma 2.3.1 and Lemma 2.3.2 on page 16).

Wellformedness of environments is defined in Figure 2.22 on page 45: it ensures that every value variable binding $x : \tau$ and equational binding $\forall(\alpha = \tau)$ depends at least on the free type variables of τ: dependencies contain syntactic dependencies. Note that dependencies are allowed to be coarser than the syntactic ones. Wellformed environments are acyclic by construction.

The setting of environments viewed as sets equipped with a dependency relation permits to use dependencies in typing rules: rules Sigma, Open and Let introduce new dependencies to keep track of cycles.

$$
\textsc{Sigma}\ \frac{\Gamma, (\forall(\alpha = \tau') \prec \mathcal{D}') \vdash M : \tau \qquad \mathcal{D}' \subseteq \mathcal{D}}{\Gamma, (\exists \beta \prec \mathcal{D}) \vdash \Sigma\ \langle\beta\rangle\ (\alpha = \tau')\, M : \tau[\beta \leftarrow \alpha]}
$$

Unsurprisingly, rule Sigma specifies that the external name β has at least all dependencies of the internal name α, among which lay the (dependencies of the) free type variables of the witness τ, as enforced by the rule Ok-Eq. This prevents the example of external recursion seen in Section 2.2.2 on page 14, which we recall below, to be welltyped:

$$
\begin{array}{l}
\{\ \ell_1 = \Sigma\ \langle\beta_1\rangle\ (\alpha_1 = \beta_2 \to \beta_2)\, M_1\ ; \\
\ \ell_2 = \Sigma\ \langle\beta_2\rangle\ (\alpha_2 = \beta_1 \to \beta_1)\, M_2\ \}
\end{array}
$$

The dependency $\beta_1 \prec \beta_2$ is required to type the first component, since the witness depends on β_2, as β_2 is a free type variable of the witness $\beta_2 \to \beta_2$. Symmetrically, $\beta_2 \prec \beta_1$ is also required to type the second component. Consequently, the zipping is forbidden because of the obvious cycle.

As opposed to the case of rule Sigma, the witness is unknown in the open construct. Hence, the condition placed on rule Open is stronger: the abstract type variable (possibly) depends on every type variable present in the context, excluding type definitions since these are only indirections: it is unnecessary to track dependencies on internal names since they are always included in the dependencies of the external names, as described by rule Sigma. Conversely, taking dependencies on internal names into account would be too coarse and impede subject reduction, since a consequence of extrusions is the expansion of the scope of internal names.

$$\textsc{Open}\quad \frac{\Gamma \vdash M : \exists \alpha.\tau \qquad \mathrm{dom}^{\forall}\,\Gamma \cup \mathrm{dom}^{\exists}\,\Gamma \subseteq \mathcal{D}}{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \mathsf{open}\ \langle \alpha \rangle\ M : \tau}$$

The previous example would again be rejected if the Σs were replaced with "open-exists" patterns, that is: the following example is rejected.

$$\begin{array}{l} \{\ \ell_1 = \mathsf{open}\ \langle \beta_1 \rangle\ \exists \gamma_1.\, \Sigma\ \langle \gamma_1 \rangle\ (\alpha_1 = \beta_2 \to \beta_2)\, M_1\ ; \\ \ \ \ell_2 = \mathsf{open}\ \langle \beta_2 \rangle\ \exists \gamma_2.\, \Sigma\ \langle \gamma_2 \rangle\ (\alpha_2 = \beta_1 \to \beta_1)\, M_2\ \} \end{array}$$

Here is an example that is well-typed, thanks to the accuracy of dependencies of Σ:

$$\begin{array}{l} \{\ \ell_1 = \Sigma\ \langle \beta_1 \rangle\ (\alpha_1 = \mathit{int})\, M_1\ ; \\ \ \ \ell_2 = \Sigma\ \langle \beta_2 \rangle\ (\alpha_2 = \beta_1 \to \beta_1)\, M_2\ \} \end{array}$$

For simplicity, let us assume that M_1 and M_2 are closed terms, that can be typed in the empty environment. Since the witness of the first branch does not depend on β_2, the term $\Sigma\ \langle \beta_1 \rangle\ (\alpha_1 = \mathit{int})\, M_1$ can be typed in the environment containing only $\exists \beta_1$. There can possibly be $\forall \beta_2$ in the environment, but β_1 does not depend on β_2. The term $\Sigma\, \langle \beta_2 \rangle\, (\alpha_2 = \beta_1 \to \beta_1)\, M_2$ can be typed in the environment containing $\forall \beta_1$ and $\exists \beta_2$, with β_2 depending on β_1. The zipping of the environments of the two branches is allowed, since there is no cycle involving β_1 and β_2.

Rewriting this piece of code with "open-exists" patterns is again well-typed, in spite of the stronger condition on rule Open, thanks to the implicit weakening in zipping: we can type the first branch without using $\forall \beta_2$ in the environment. Therefore, the condition $\beta_1 \prec \beta_2$ is not required by rule Open in the first branch and no cycle is detected.

Finally, rule Let highlights variables that are used, hence possibly hidden in an existential value, in the first branch of the let and used in an opening in the second branch: these variables belong to $\mathrm{dom}^{\forall}\,\Gamma_1 \cap \mathrm{dom}^{\exists}\,\Gamma_2$.

$$\textsc{Let}\quad \frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2 \qquad \mathrm{dom}^{\forall}\,\Gamma_1 \cap \mathrm{dom}^{\exists}\,\Gamma_2 \subseteq \mathcal{D}}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

Therefore, the value variable x that is bound in the let must depend on the variables in $\mathrm{dom}^{\forall}\,\Gamma_1 \cap \mathrm{dom}^{\exists}\,\Gamma_2$. These are indeed responsible for the cycle in the example of internal recursion seen in Section 2.2.2 on page 14 and reproduced below:

$$\mathsf{let}\ x = \exists(\alpha = \beta \to \beta)\, M\ \mathsf{in}\ \mathsf{open}\ \langle \beta \rangle\ x$$

The binding $\forall \beta$ is required in the typing environment of the bound expression $\exists(\alpha = \beta \to \beta)\, M$, whereas the binding $\exists \beta$ appears in the typing environment for the body $\mathsf{open}\ \langle \beta \rangle\ x$. Thus, the constraint $x \prec \beta$ is required in the typing environment of $\mathsf{open}\ \langle \beta \rangle\ x$, which fails, as rule Open requests that $\exists \beta$ must be minimal in the dependency relation. Notice that, thanks to weakening, the

next piece of program would be accepted, as long as x does not occur free in M':

$$\mathsf{let}\ x = \exists(\alpha = \beta \to \beta)\, M\ \mathsf{in}\ \{\ell_1 = \mathsf{open}\ \langle\beta\rangle\ M'\ ;\ \ell_2 = x\}$$

Here, the record $\{\ell_1 = \mathsf{open}\ \langle\beta\rangle\ M'\ ;\ \ell_2 = x\}$ is typed in an environment that contains the bindings $\exists\beta$ and $x : \tau$, and that mentions $x \prec \beta$. But $\mathsf{open}\ \langle\beta\rangle\ M'$ can be typed in an environment that contains $\exists\beta$ but does not contain x. Moreover, x can be typed in an environment containing $\forall\beta$ and $x : \tau$. Thus, the zipping of the two environments of the two components of the record is allowed, so the whole term is accepted.

2.6.3 Double vision

Defining an expression that manipulates an abstract type before its witness has been given is sometimes desirable, as it brings more freedom in the code structure. It may also become necessary when building recursive values. Currently, the following term is considered as ill-typed:

$$\exists\beta.\, \mathsf{let}\ f = \lambda(x : \beta)\, x\ \mathsf{in}\ \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\, \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = f\}$$

This is because rule Sigma (page 11) does not let the external name β be visible in its premise, hence f is not allowed to occur under the Σ. It is easy to correct this by leaving a $\forall\beta$ in the premise instead of $\exists\beta$ (see the rule below). However, the following piece of code would still be rejected:

$$\exists\beta.\, \mathsf{let}\ f = \lambda(x : \beta)\, x\ \mathsf{in}\ \Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})\, f\ (1 : \alpha)$$

After the existential resource β is introduced, it defines f as the identity on β and then uses f in the context of the open witness definition $\Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int})$. However, we do not know that α and β denote the *same* witness: the application $f\ (1 : \alpha)$ is ill-typed.

This is called the *double vision problem*: it characterizes the inability to maintain a link between the internal and external view of a given type. This problem is well-known in the study of recursive modules, but as we can see it already happens in the absence of recursion. To solve this problem, it suffices to carry the missing information in the context (for clarity, dependencies are omitted):

$$\textsc{Sigma}\ \frac{\Gamma, \forall\beta, \Gamma', \forall(\alpha \lhd \beta = \tau') \vdash M : \tau}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\ \langle\beta\rangle\ (\alpha = \tau')\, M : \tau[\alpha \leftarrow \beta]}$$

The typing environment is enriched with a new kind of equation $\forall(\alpha \lhd \beta = \tau')$, which says that the witness τ' is denoted by the internal name α, and, in addition, that the external name β can be viewed internally as α. This is realized through the use of the *similarity* relation defined under a context Γ and written $\lhd$ that satisfies all the equalities between internal and external names that are present in the context Γ. It is used through rule Sim.

$$\textsc{Sim}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash M : \tau}$$

One may wonder why we decided to use both an external and an internal name: they indeed denote the same object. In RTG, a single type reference is used along with two scopes and only one of them contains a type definition. We give two reasons for handling two names and an equation relating them: first, it corresponds to practice in recursive modules, where a single type component is reached through two different paths, which leads to the double vision problem. Second, the use of two names makes programs more maintainable in the sense that it is more respectful to the notion

of *interface*: whatever the choice of internal name, the external name can remain fixed. Thus, one can apply an internal renaming without changing the external type.

2.6.4 Recursive types

Extended non-recursive type definitions in Section 2.6.2 and Section 2.6.3 led to a finer type checking but did not require a change in the semantics. By contrast, permitting recursive type definitions has the reverse effect: typing is nearly unchanged, but semantics must be adapted.

We extend the type algebra with a fixpoint and specify with the use of the symbol $\approx$ instead of $=$ when a type equation is allowed to contribute to a cycle. Type wellformedness (Figure 2.23 on page 45) forbids the body of a recursive type to be a type variable. This syntactic requirement ensures the contractiveness of recursive types. Contractiveness is required to keep the equational theory on recursive types sound.

$$\begin{array}{llll} \simeq & ::= & = & \mid \approx \\ \tau & ::= & \ldots & \mid \mu\alpha.\tau \\ M & ::= & \ldots & \mid \Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,M \\ w & ::= & \ldots & \mid \Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,w \end{array}$$

We also change the type coercibility relation to cope with recursive types: we *coinductively* define the relation (see Figure 2.24 on page 45), so that it is symmetric and transitive, and includes the usual *unfolding* rules for recursive types (Coerce-Fix-Left and Coerce-Fix-Right). The unfolding rules are guarded so that they can only be used on a recursive type whose body is not a variable. The rules for unfolding equations (Coerce-Eq-Left and Coerce-Eq-Right) follow the same style, as well as the rules that equate internal and external names (Coerce-Sim-Left and Coerce-Sim-Right). Every rule is defined so that it is productive.

The typing rule Sigma is also changed (Figure 2.20 on page 44): the dependencies are tracked only for a non-recursive witness definition. As a consequence, any cycle of dependencies can pass through a given recursive witness definition.

Then, we add the following rules to the reduction relation:

$$\Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau)\,\Sigma\,\langle\beta'\rangle\,(\alpha' \simeq \tau')\,w \longrightarrow \Sigma\,\langle\beta'\rangle\,(\alpha' \simeq \tau'[\alpha \leftarrow \beta])\,\Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau)\,w$$

$$\begin{array}{l} \nabla\beta'.\,\Sigma\,\langle\beta'\rangle\,(\alpha' \approx \tau')\,(\Sigma\,\langle\beta_i\rangle\,(\alpha_i \simeq \tau_i)\,)^{i\in I}\,\nu \\ \quad \longrightarrow \nabla\beta'.\,\Sigma\,\langle\beta'\rangle\,(\alpha' = \mathrm{close}(\alpha' \triangleleft \beta' \approx \tau', (\alpha_i \triangleleft \beta_i \simeq \tau_i)^{i\in I}))\,(\Sigma\,\langle\beta_i\rangle\,(\alpha_i \simeq \tau_i)\,)^{i\in I}\,\nu \\ \quad \text{where } \nabla \text{ stands for } \nu \text{ or } \exists \end{array}$$

When two Σs have to be exchanged, it is no longer possible to substitute the first witness into the second one for wellformedness reasons. Instead, we replace the first internal name with the external one during swapping, as described by the first reduction rule. The second rule specifies that a closed or restricted, potentially recursive type definition can be resolved into a non-recursive one, that involves a recursive witness. To do this, the close operator, that is defined in Figure 2.19 on the next page, gathers the list of the other witnesses and ties the recursive knot. Thanks to co-induction, the provable equalities are unchanged by the closure.

The reduction below exemplifies the closure operation:

$$\begin{array}{lll} & \nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 \approx \alpha_1 \times \beta_2)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \alpha_1 \times \alpha_2)\,\nu & \\ \longrightarrow & \nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \alpha_1 \times \alpha_2)\,\nu & \text{by fixpoint closure} \\ \longrightarrow & \nu\beta_1.\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \tau \times \alpha_2)\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau)\,\nu & \text{by exchange of } \Sigma\text{s} \\ \longrightarrow & \Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \tau \times \alpha_2)\,\nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau)\,\nu & \text{by Sigma-Nu} \\ \longrightarrow & \Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \tau \times \alpha_2)\,\nu[\alpha_1 \leftarrow \tau] & \text{by Nu-Sigma} \end{array}$$

$$
\begin{array}{rcl}
\mathrm{close}(\alpha \lhd \beta = \tau) & \triangleq & \tau \\
\mathrm{close}(\alpha \lhd \beta \approx \tau) & \triangleq & \mu\alpha.\tau \\
\mathrm{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i)^{i\in I}, \alpha' \lhd \beta' = \tau') & \triangleq & \mathrm{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i[\beta' \leftarrow \tau'[\beta_j \leftarrow \alpha_j]])^{i\in I}) \\
\mathrm{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i)^{i\in I}, \alpha' \lhd \beta' \approx \tau') & \triangleq & \mathrm{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i[\beta' \leftarrow \mu\alpha'.\tau'[\beta_j \leftarrow \alpha_j]])^{i\in I})
\end{array}
$$

Figure 2.19: Closing mutually recursive type equations.

where $\tau = \mathrm{close}(\alpha_1 \lhd \beta_1 \approx \alpha_1 \times \beta_2, \alpha_2 \lhd \beta_2 \approx \alpha_1 \times \alpha_2) = \mu\alpha_1.(\alpha_1 \times \mu\alpha_2.(\alpha_1 \times \alpha_2))$.

The term we consider contains two mutually recursive type definitions, and the external name β_1 of the first one is restricted. The close operator computes the closed witness τ, which becomes the new, recursive witness of β_1, defined by a non-recursive equation. Then, the innermost Σ can be extruded, and the restricted equation is eventually eliminated.

By definition, this semantics ensures that only equations that are marked as potentially recursive may actually create recursive types during reduction. Type soundness ensures that this is sufficient to reduce well-typed programs, *i.e.* that recursive types are never needed in other configurations. Hence, although abstract types can be used in a flexible manner, the risk of inadvertently using recursive types via type abstraction can be tracked by the type system and finely tuned by the user.

It is also interesting that mutually recursive equations are explicitly resolved during reduction, and moreover in a standard way.

2.6.5 Recursive values

In this section, we extend $\mathsf{F}^{\forall}$ with recursive values of the form $\mu(x:\tau)\,v$, which are necessary to model recursive modules. Although it is possible to use the well-known backpatching semantics for fixpoints [Dre07a, Dre04, Bou04], we prefer a storeless, unrolling-based semantics, so as to avoid the need for references.

Our unrolling semantics lies between the backpatching semantics, which computes recursive values at their creation and fails if they are ill-founded, and the lazy semantics, which unfolds recursive values only at their use. As the former, we evaluate recursive definitions at their creation, by letting evaluation proceed under fixpoints, but *without* unrolling them. Instead, fixpoints are unrolled *on demand* when they need to be destructed, as with the lazy semantics. (As with the lazy semantics, ill-founded recursion may thus loop at its use instead of its creation.) The two aspects of our semantics are captured by the form of evaluation contexts and the following reduction rules:

$$
\begin{array}{l}
\mathsf{E} \quad ::= \quad \ldots \quad | \quad \mu(x:\tau)\,\mathsf{E} \\
\mu(x:\tau)\,\Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau')\,w \longrightarrow \Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau')\,\mu(x:\tau)\,w \qquad \text{when } \alpha \notin \mathrm{ftv}(\tau) \\
\mathsf{R}[\mu(x:\tau)\,v] \longrightarrow \mathsf{R}[\mathsf{let}\ x = \mu(x:\tau)\,v\ \mathsf{in}\ v]
\end{array}
$$

where R is a blocked redex-form, that is, an application $[\cdot]\,v$, an instantiation $[\cdot]\,\tau$, a projection $[\cdot].\ell$, or an opening $\mathsf{open}\,\langle\alpha\rangle\,[\cdot]$. Remark that $\mathsf{let}\ x = \mu(y:\tau)\,v\ \mathsf{in}\ M$ is not a blocked term, because the reduction rule for let has no other restriction on its left subterm than being a value. The redex $\nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mu(x:\tau')\,v$ is not blocked either, for the same reason. These definitions allow evaluation to proceed under fixpoints until one gets a result, then extrusion can proceed through fixpoints to obtain a recursive value, which will be expanded on demand, when obstructed by a redex-form.

In order to enable unrolling, one must ensure that reducing under fixpoints and extruding Σs always give rise to a value, because impure results cannot be substituted. For this purpose, we restrict the body of fixpoints to be *extended results*, denoted by s, which are either results or themselves

records, let-bindings, or projections of extended results. The reason of this restriction is to be able to extract the sources of linearity from the body of fixpoints, so that they can be unfolded without breaking the linearity invariant.

$$\begin{array}{lcll} M & ::= & \ldots \mid \mu(x:\tau)\,s & \text{(Terms)} \\ v & ::= & \ldots \mid \mu(x:\tau)\,v \mid p & \text{(Values)} \\ p & ::= & x \mid p.\ell & \text{(Paths)} \\ s & ::= & w \mid \mathsf{let}\ x = s\ \mathsf{in}\ s \mid \{(\ell_i = s_i)^i\} \mid s.\ell & \text{(Extended results)} \end{array}$$

Values are extended with both fixpoints of values and paths, *i.e.* projections built from variables. The reason for adding paths to the class of values is to allow a lazy semantics of fixpoints in a setting where recursion is not guarded: this allows for instance the term $\mu(x:\{\ell_1:\tau\ ;\ \ell_2:\tau\}\{\ell_1 = x.\ell_2\ ;\ell_2 = x.\ell_1\})$ to be considered as a value, that will be unfolded once, if it is projected. Note that we disallow applications that are not guarded by λs or Λs, because otherwise we could write terms like $\mathsf{let}\ f = \lambda(x:\exists\alpha.\tau)\,\exists\alpha.\,\mathsf{open}\ \langle\alpha\rangle\ x\ \mathsf{in}\ \mu(x:\exists\alpha.\tau)\,f\,x$, which reduces to $\mu(x:\exists\alpha.\tau)\,\exists\alpha.\,\mathsf{open}\ \langle\alpha\rangle\ x$ and does not respect the simple syntactic criteria on bodies of fixpoints. This restriction permits to avoid the strongly related problems of reducing arbitrary terms containing free term variables and of defining *strong values*, *i.e.* values that contain redexes that are blocked by the presence of variables. Such values appear when one considers strong reduction, *i.e.* reducing under λs and Λs. The problem of strong reduction is discussed in Section 2.8.1 on page 52.

Adding term-level recursion to $\mathsf{F}^{\curlyvee}$ is not an issue: indeed, this is a direct consequence of the modularity of the constructs for open existential types. The only requirement is to permit the evaluation under fixpoints just enough to get a pure term: extrusion of Σs throught fixpoints is essential. A benefit of our approach is that it permits to keep a standard style of presentation: it uses evaluation contexts, and avoids using references to model recursion, as in the *backpatching* semantics for fixpoints.

2.6.6 Soundness of the extensions

We now review the main lemmas on which the soundness proof of the extended type system is built. The structure of the proof is the same as for the type soudness of Core $\mathsf{F}^{\curlyvee}$ (Section 2.4.5). We first define a new refinement on domain of contexts: $\mathrm{dom}^{\lhd}\,\Gamma = \{\beta \mid \forall(\alpha \lhd \beta \simeq \tau) \in \Gamma\}$.

Lemmas about regularity

The next lemma permits better inductive reasoning on judgments that involve environments.

Lemma 2.6.2. *Assume $\mathcal{R}$ is a transitive anti-reflexive relation over a finite set $\mathcal{E}$. Then there exists a relation $\hat{\mathcal{R}}$ that is compatible with $\mathcal{R}$, transitive, anti-reflexive,* and *total.*

Proof. By induction on the cardinality of $\mathcal{E}$, using the fact that there exists a minimal element for $\mathcal{R}$ (but it is generally not necessarily the *least* one). □

For instance, when reasoning about environment wellformedness, instead of proceeding by induction on the judgment itself, which imposes an arbitrary ordering on bindings, one can reason on any ordering of the binders that is compatible with de dependency relation of the environment.

The next two lemmas ensure that wellformed environment are acyclic, and that zipping is compatible with wellformedness.

Lemma 2.6.3. *The following assertions hold:*

- *If* $\Gamma \vdash \mathrm{ok}$ *then* Γ *is acyclic;*
- *If* $\Gamma \vdash \tau :: \star$ *then* Γ *is acyclic.*

$$\textsc{Var}\ \frac{\Gamma\,\mathrm{pure} \qquad \Gamma \vdash \mathrm{ok}}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{Lam}\ \frac{\Gamma\,\mathrm{pure}}{\Gamma, (x : \tau_1 \prec \mathcal{D}) \vdash M : \tau_2} \over \Gamma \vdash \lambda(x : \tau_1)\, M : \tau_1 \to \tau_2 \qquad \textsc{App}\ \frac{\Gamma_1 \vdash M_1 : \tau_2 \to \tau \qquad \Gamma_2 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2 : \tau}$$

$$\textsc{Let}\ \frac{\mathrm{dom}^{\forall}\,\Gamma_1 \cap \mathrm{dom}^{\exists}\,\Gamma_2 \subseteq \mathcal{D} \qquad \Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2} \qquad \textsc{Gen}\ \frac{\Gamma\,\mathrm{pure}}{\Gamma, (\forall\alpha \prec \mathcal{D}) \vdash M : \tau} \over \Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau \qquad \textsc{Inst}\ \frac{\Gamma \vdash M : \forall\alpha.\tau' \qquad \Gamma \vdash \tau :: \star}{\Gamma \vdash M\,\tau : \tau'[\alpha \leftarrow \tau]}$$

$$\textsc{Empty}\ \frac{\Gamma\,\mathrm{pure} \qquad \Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} : \{\}} \qquad \textsc{Record}\ \frac{(\Gamma_i \vdash M_i : \tau_i)^{i \in 1..n} \qquad \mathsf{injective}\,(i \mapsto \ell_i)^{i \in 1..n}}{\Gamma_1 \curlyvee \cdots \curlyvee \Gamma_n \vdash \{(\ell_i = M_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}}$$

$$\textsc{Proj}\ \frac{\Gamma \vdash M : \{(\ell_i : \tau_i)^{i \in 1..n}\} \qquad 1 \leqslant k \leqslant n}{\Gamma \vdash M.\ell_k : \tau_k} \qquad \textsc{Exists}\ \frac{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M : \tau}{\Gamma \vdash \exists\alpha.\, M : \exists\alpha.\tau} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

$$\textsc{Sigma}\ \frac{\mathcal{D}' \setminus \{\beta\} \subseteq \mathcal{D},\ \text{if } \simeq \text{ is } = \qquad \Gamma, (\forall\beta \prec \mathcal{D}), \Gamma', (\forall(\alpha \lhd \beta \simeq \tau') \prec \mathcal{D}') \vdash M : \tau}{\Gamma, (\exists\beta \prec \mathcal{D}), \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau')\, M : \tau[\alpha \leftarrow \beta]} \qquad \textsc{Open}\ \frac{\Gamma \vdash M : \exists\alpha.\tau \qquad \mathrm{dom}^{\forall}\,\Gamma \cup \mathrm{dom}^{\exists}\,\Gamma \subseteq \mathcal{D}}{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash \mathsf{open}\,\langle\alpha\rangle\, M : \tau}$$

$$\textsc{Nu}\ \frac{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M : \tau \qquad \alpha \notin \mathrm{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\, M : \tau} \qquad \textsc{Weaken}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma' \sqsupseteq \Gamma}{\Gamma' \vdash M : \tau} \qquad \textsc{Sim}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash M : \tau}$$

$$\textsc{Fix}\ \frac{\Gamma, (x : \tau \prec \mathcal{D}) \vdash s : \tau}{\Gamma \vdash \mu(x : \tau)\, s : \tau}$$

Figure 2.20: Typing rules of the extended system.

$$\textsc{Entail-Refl}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \sqsupseteq \Gamma} \qquad \textsc{Entail-Trans}\ \frac{\Gamma_1 \sqsupseteq \Gamma_2 \qquad \Gamma_2 \sqsupseteq \Gamma_3}{\Gamma_1 \sqsupseteq \Gamma_3} \qquad \textsc{Entail-Binding}\ \frac{\Gamma, (b \prec \mathcal{D}) \vdash \mathrm{ok} \qquad b \neq \exists\alpha}{\Gamma, (b \prec \mathcal{D}) \sqsupseteq \Gamma}$$

Figure 2.21: Weaker environments.

$$\textsc{Ok-Var}\ \frac{\mathrm{ftv}(\tau) \subseteq \mathcal{D} \subseteq \mathrm{dom}\,\Gamma \quad \Gamma \vdash \tau :: \star \quad x \notin \mathrm{dom}\,\Gamma}{\Gamma, (x : \tau \prec \mathcal{D}) \vdash \mathrm{ok}} \qquad \textsc{Ok-Exists}\ \frac{\mathcal{D} \subseteq \mathrm{dom}\,\Gamma \quad \Gamma \vdash \mathrm{ok} \quad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \mathrm{ok}}$$

$$\textsc{Ok-Eq}\ \frac{\mathrm{ftv}(\tau) \subseteq \mathcal{D} \subseteq \mathrm{dom}\,\Gamma \quad \forall \beta \in \Gamma \quad \Gamma \vdash \tau :: \star \quad \alpha \notin \mathrm{dom}\,\Gamma \quad \beta \notin \mathrm{dom}^{\lhd}\,\Gamma}{\Gamma, (\forall (\alpha \lhd \beta = \tau) \prec \mathcal{D} \uplus \{\beta\}) \vdash \mathrm{ok}}$$

$$\textsc{Ok-Forall}\ \frac{\mathcal{D} \subseteq \mathrm{dom}\,\Gamma \quad \Gamma \vdash \mathrm{ok} \quad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \mathrm{ok}} \qquad \textsc{Ok-EqRec}\ \frac{\mathrm{ftv}(\mu\alpha.\tau) \subseteq \mathcal{D} \subseteq \mathrm{dom}\,\Gamma \quad \forall \beta \in \Gamma \quad \Gamma \vdash \mu\alpha.\tau :: \star \quad \alpha \notin \mathrm{dom}\,\Gamma \quad \beta \notin \mathrm{dom}^{\lhd}\,\Gamma}{\Gamma, (\forall (\alpha \lhd \beta \approx \tau) \prec \mathcal{D} \cup \{\beta\}) \vdash \mathrm{ok}} \qquad \textsc{Ok-Empty}\ \frac{}{\varepsilon \vdash \mathrm{ok}}$$

Figure 2.22: Wellformed environments.

$$\textsc{Wf-Var}\ \frac{\Gamma \vdash \mathrm{ok} \quad \alpha \in \mathrm{dom}\,\Gamma}{\Gamma \vdash \alpha :: \star} \qquad \textsc{Wf-Arrow}\ \frac{\Gamma \vdash \tau_1 :: \star \quad \Gamma \vdash \tau_2 :: \star}{\Gamma \vdash \tau_1 \to \tau_2 :: \star} \qquad \textsc{Wf-Record}\ \frac{\mathsf{injective}\ (i \mapsto \ell_i)^{i \in 1..n} \quad (\Gamma \vdash \tau_i :: \star)^{i \in 1..n}}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} :: \star} \qquad \textsc{Wf-Empty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} :: \star}$$

$$\textsc{Wf-Forall}\ \frac{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \tau :: \star}{\Gamma \vdash \forall \alpha.\tau :: \star} \qquad \textsc{Wf-Exists}\ \frac{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \tau :: \star}{\Gamma \vdash \exists \alpha.\tau :: \star} \qquad \textsc{Wf-Mu}\ \frac{\tau \text{ is not a variable} \quad \Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \tau :: \star}{\Gamma \vdash \mu\alpha.\tau :: \star}$$

Figure 2.23: Wellformed types.

$$\textsc{Coerce-Refl}\ \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \tau \sim \tau} \qquad \textsc{Coerce-Eq-Left}\ \frac{\Gamma \vdash \tau \sim \tau' \quad \forall (\alpha \lhd \beta \simeq \tau) \in \Gamma}{\Gamma \vdash \alpha \sim \tau'} \qquad \textsc{Coerce-Eq-Right}\ \frac{\Gamma \vdash \tau \sim \tau' \quad \forall (\alpha \lhd \beta \simeq \tau') \in \Gamma}{\Gamma \vdash \tau \sim \alpha} \qquad \textsc{Coerce-Sim-Left}\ \frac{\Gamma \vdash \alpha \sim \tau \quad \forall (\alpha \lhd \beta \simeq \tau') \in \Gamma}{\Gamma \vdash \beta \sim \tau}$$

$$\textsc{Coerce-Sim-Right}\ \frac{\Gamma \vdash \tau \sim \alpha \quad \forall (\alpha \lhd \beta \simeq \tau') \in \Gamma}{\Gamma \vdash \tau \sim \beta} \qquad \textsc{Coerce-Fix-Left}\ \frac{\tau \text{ is not a variable} \quad \Gamma \vdash \tau[\alpha \leftarrow \mu\alpha.\tau] \sim \tau'}{\Gamma \vdash \mu\alpha.\tau \sim \tau'} \qquad \textsc{Coerce-Fix-Right}\ \frac{\tau \text{ is not a variable} \quad \Gamma \vdash \tau' \sim \tau[\alpha \leftarrow \mu\alpha.\tau]}{\Gamma \vdash \tau' \sim \mu\alpha.\tau}$$

$$\textsc{Coerce-Arrow}\ \frac{\Gamma \vdash \tau_1 \sim \tau_1' \quad \Gamma \vdash \tau_2 \sim \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

$$\textsc{Coerce-Record}\ \frac{(\Gamma \vdash \tau_i \sim \tau_i')^{i \in 1..n} \quad \mathsf{injective}\ (i \mapsto \ell_i)^{i \in 1..n}}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \sim \{(\ell_i : \tau_i')^{i \in 1..n}\}} \qquad \textsc{Coerce-Forall}\ \frac{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \tau \sim \tau'}{\Gamma \vdash \forall \alpha.\tau \sim \forall \alpha.\tau'} \qquad \textsc{Coerce-Exists}\ \frac{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \tau \sim \tau'}{\Gamma \vdash \exists \alpha.\tau \sim \exists \alpha.\tau'}$$

Figure 2.24: Coercible types (co-inductive definition).

$$\textsc{Sim-Refl}\ \frac{\Gamma \vdash \mathrm{ok} \quad \alpha \in \mathrm{dom}\,\Gamma}{\Gamma \vdash \alpha \lhd \alpha} \qquad \textsc{Sim-Eq}\ \frac{\Gamma \vdash \mathrm{ok} \quad \forall(\alpha \lhd \beta = \tau) \in \Gamma}{\Gamma \vdash \alpha \lhd \beta} \qquad \textsc{Sim-Empty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} \lhd \{\}}$$

(Rules for symmetry, transitivity and congruence are omitted.)

Figure 2.25: Similar types.

Proof. The proof proceeds by mutual induction on the judgments. The key argument is the fact that only bindings that are minimal for the dependency are added to the environment. □

Lemma 2.6.4. *Assume that* $\Gamma_2 \vdash \mathrm{ok}$ *and* $\Gamma_1 \curlyvee \Gamma_2$ *is well defined. The following assertions hold:*

- *If* $\Gamma_1 \vdash \mathrm{ok}$*, then* $\Gamma_1 \curlyvee \Gamma_2 \vdash \mathrm{ok}$*;*
- *If* $\Gamma_1 \vdash \tau :: \star$*, then* $\Gamma_1 \curlyvee \Gamma_2 \vdash \tau :: \star$*.*

Proof. By mutual induction on the judgments. Without loss of generality, thanks to Lemma 2.6.2 on page 43, we can assume that the wellformedness proofs for environments Γ_1 and Γ_2 are done in an order that is compatible with the dependencies of $\Gamma_1 \curlyvee \Gamma_2$. □

The other lemmas ensure that the parameters of every judgment are all wellformed.

Lemma 2.6.5. *Assume* $\Gamma_1 \vdash \tau_1 :: \star$ *and* $\mathrm{ftv}(\tau_1) \subseteq \mathcal{D}$ *and* $\alpha \notin \mathrm{dom}^{\lhd}\,\Gamma$*. The following assertions hold:*

- *If* $\Gamma_1, (\forall \alpha \prec \mathcal{D}), \Gamma_2 \vdash \mathrm{ok}$*, then* $\Gamma_1, \Gamma_2'[\alpha \leftarrow \tau_1] \vdash \mathrm{ok}$*;*
- *If* $\Gamma_1, (\forall \alpha \prec \mathcal{D}), \Gamma_2 \vdash \tau_2 :: \star$*, then* $\Gamma_1, \Gamma_2'[\alpha \leftarrow \tau] \vdash \tau_2[\alpha \leftarrow \tau_1] :: \star$*.*

Proof. By mutual induction. □

Lemma 2.6.6. *If* $\Gamma \vdash \mu\alpha.\tau :: \star$*, then* $\Gamma \vdash \tau[\alpha \leftarrow \mu\alpha.\tau] :: \star$*.*

Proof. By inversion of the type wellformedness judgment, then using lemma Lemma 2.6.5. □

Lemma 2.6.7. *If* $\Gamma \vdash \tau[\alpha \leftarrow \mu\alpha.\tau] :: \star$*, then* $\Gamma \vdash \mu\alpha.\tau :: \star$*.*

Proof. By induction on τ and inversion of the typing judgment. □

Lemma 2.6.8. *If* $\Gamma \vdash \tau_1 \lhd \tau_2$*, then* $\Gamma \vdash \tau_1 :: \star$ *and* $\Gamma \vdash \tau_2 :: \star$*.*

Proof. By induction on the similarity judgment. □

Lemma 2.6.9. *If* $\Gamma \vdash \tau_1 \sim \tau_2$*, then* $\Gamma \vdash \tau_1 :: \star$ *and* $\Gamma \vdash \tau_2 :: \star$*.*

Proof. By coinduction on the equivalence judgment, using Lemma 2.6.6 and Lemma 2.6.7. □

Lemma 2.6.10. *If* $\Gamma_1 \sqsupseteq \Gamma_2$*, then* $\Gamma_1 \vdash \mathrm{ok}$ *and* $\Gamma_2 \vdash \mathrm{ok}$*.*

Proof. By induction on the weakening judgment. □

Lemma 2.6.11. *If* $\Gamma \vdash M : \tau$*, then* $\Gamma \vdash \tau :: \star$*.*

Proof. By induction on the typing judgment, using other lemmas about regularity (Lemma 2.6.4, Lemma 2.6.9, Lemma 2.6.8 and Lemma 2.6.10). □

Lemmas about substitution

This section gathers lemmas necessary to prove the soundness of substitution of terms.

Lemma 2.6.12. *The following assertions hold:*

- *If* $\Gamma_1, (x : \tau \prec \mathcal{D}), \Gamma_2 \vdash \mathrm{ok}$, *then* $\Gamma_1, \Gamma_2 \vdash \mathrm{ok}$;
- *If* $\Gamma_1, (x : \tau \prec \mathcal{D}), \Gamma_2 \vdash \tau' :: \star$, *then* $\Gamma_1, \Gamma_2 \vdash \tau :: \star$.

Lemma 2.6.13. *If* $\Gamma_1, (x : \tau \prec \mathcal{D}), \Gamma_2 \vdash \tau_1 \sim \tau_2$, *then* $\Gamma_1, \Gamma_2 \vdash \tau_1 \sim \tau_2$.

Lemma 2.6.14. *If* $\Gamma_1, (x : \tau \prec \mathcal{D}), \Gamma_2 \vdash \tau_1 \lhd \tau_2$, *then* $\Gamma_1, \Gamma_2 \vdash \tau_1 \lhd \tau_2$.

Lemma 2.6.15. *If* $\Gamma_1, (x : \tau \prec \mathcal{D}), \Gamma_2 \vdash M_2 : \tau_2$ *and* $\Gamma_1' \vdash M_1 : \tau_1$ *and* $\Gamma_1'\,\mathrm{pure}$ *and* $\Gamma_1 \curlyvee \Gamma_1'$ *is well defined and* $\mathrm{dom}\,\Gamma_1' \cap \mathrm{dom}\,\Gamma_2 = \emptyset$, *then* $(\Gamma_1 \curlyvee \Gamma_1'), \Gamma_2 \vdash M_2[x \leftarrow M_1] : \tau_2$.

Lemmas about instantiation

This section gathers lemmas necessary to prove the soundness of substitution of types.

Lemma 2.6.16. *If* $\Gamma_1, (\forall \alpha \prec \mathcal{D}), \Gamma_2 \vdash \tau \sim \tau'$ *and* $\Gamma_1 \vdash \tau_1 :: \star$ *and* $\mathrm{ftv}(\tau_1) \subseteq \mathcal{D}$ *and* $\alpha \notin \mathrm{dom}^{\lhd}\,\Gamma_2$, *then* $\Gamma_1, \Gamma_2[\alpha \leftarrow \tau_1] \vdash \tau[\alpha \leftarrow \tau_1] \sim \tau'[\alpha \leftarrow \tau_1]$.

Lemma 2.6.17. *If* $\Gamma_1, (\forall \alpha \prec \mathcal{D}), \Gamma_2 \vdash \tau \lhd \tau'$ *and* $\Gamma_1 \vdash \tau_1 :: \star$ *and* $\mathrm{ftv}(\tau_1) \subseteq \mathcal{D}$ *and* $\alpha \notin \mathrm{dom}^{\lhd}\,\Gamma_2$, *then* $\Gamma_1, \Gamma_2[\alpha \leftarrow \tau_1] \vdash \tau[\alpha \leftarrow \tau_1] \lhd \tau'[\alpha \leftarrow \tau_1]$.

Lemma 2.6.18. *If* $\Gamma_1, (\forall \alpha \prec \mathcal{D}), \Gamma_2 \vdash M : \tau$ *and* $\Gamma_1 \vdash \tau_1 :: \star$ *and* $\mathrm{ftv}(\tau_1) \subseteq \mathcal{D}$ *and* $\alpha \notin \mathrm{dom}^{\lhd}\,\Gamma_2$, *then* $\Gamma_1, \Gamma_2[\alpha \leftarrow \tau_1] \vdash M[\alpha \leftarrow \tau_1] : \tau[\alpha \leftarrow \tau_1]$.

Lemmas about swapping of equations

This section gathers lemmas necessary to prove the soundness of swapping of witness definitions. They state that for every judgment, swapping two witness definitions in the environment is valid.

Lemma 2.6.19. *The following assertions hold:*

- *If* $\Gamma_1, (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2) \prec \mathcal{D}_2), \Gamma_2 \vdash \mathrm{ok}$ *holds, then* $\Gamma_1, (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2[\alpha_1 \leftarrow \beta_1]) \prec \mathcal{D}_2 \setminus \{\alpha_1\}), (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), \Gamma_2 \vdash \mathrm{ok}$ *holds;*
- *If* $\Gamma_1, (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2) \prec \mathcal{D}_2), \Gamma_2 \vdash \tau :: \star$ *holds, then* $\Gamma_1, (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2[\alpha_1 \leftarrow \beta_1]) \prec \mathcal{D}_2 \setminus \{\alpha_1\}), (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), \Gamma_2 \vdash \tau :: \star$ *holds.*

Lemma 2.6.20. *If* $\Gamma_1, (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2) \prec \mathcal{D}_2), \Gamma_2 \vdash \tau \sim \tau'$ *holds, then* $\Gamma_1, (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2[\alpha_1 \leftarrow \beta_1]) \prec \mathcal{D}_2 \setminus \{\alpha_1\}), (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), \Gamma_2 \vdash \tau \sim \tau'$ *holds.*

Lemma 2.6.21. *If* $\Gamma_1, (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2) \prec \mathcal{D}_2), \Gamma_2 \vdash \tau \lhd \tau'$ *holds, then* $\Gamma_1, (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2[\alpha_1 \leftarrow \beta_1]) \prec \mathcal{D}_2 \setminus \{\alpha_1\}), (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), \Gamma_2 \vdash \tau \lhd \tau'$ *holds.*

Lemma 2.6.22. *If* $\Gamma_1, (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2) \prec \mathcal{D}_2), \Gamma_2 \vdash M : \tau$ *holds, then* $\Gamma_1, (\forall(\alpha_2 \lhd \beta_2 \simeq \tau_2[\alpha_1 \leftarrow \beta_1]) \prec \mathcal{D}_2 \setminus \{\alpha_1\}), (\forall(\alpha_1 \lhd \beta_1 \simeq \tau_1) \prec \mathcal{D}_1), \Gamma_2 \vdash M : \tau$ *holds.*

Lemmas about fixpoint closure

The next lemmas cope with the soundness of fixpoint closure. The first two lemmas deal with the replacement of a witness in the environment with an equivalent type.

Lemma 2.6.23. *If* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau \sim \tau'$ *and* $\Gamma_1 \vdash \tau' :: \star$*, then the following assertions hold:*

- $\Gamma_1, (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}), \Gamma_2 \vdash \mathrm{ok}$*;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 :: \star$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 :: \star$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \lhd \tau_0'$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \lhd \tau_0'$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \sim \tau_0'$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \sim \tau_0'$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D}), \Gamma_2 \vdash M : \tau_0$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau : \tau_0$ *holds.*

Lemma 2.6.24. *If* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau \sim \tau'$ *and* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}) \vdash \tau' :: \star$ *and* τ' *is not a variable, then the following assertions hold:*

- $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau') \prec \mathcal{D}), \Gamma_2 \vdash \mathrm{ok}$*;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 :: \star$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 :: \star$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \lhd \tau_0'$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \lhd \tau_0'$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \sim \tau_0'$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau_0 \sim \tau_0'$ *holds;*
- *if* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D}), \Gamma_2 \vdash M : \tau_0$*, then* $\Gamma_1, (\forall(\alpha \lhd \beta \approx \tau') \prec \mathcal{D}), \Gamma_2 \vdash \tau : \tau_0$ *holds.*

Lemma 2.6.25. *Assume* $\Gamma \vdash \mathrm{ok}$ *and* $\forall(\alpha \lhd \beta \approx \tau) \in \Gamma$*. Then* $\Gamma \vdash \alpha \sim \mu\alpha.\tau[\beta \leftarrow \alpha]$ *holds.*

Lemma 2.6.26. *Let* $\Gamma' = \Gamma, (\forall(\alpha_i \lhd \beta_i \simeq \tau_i))^{i \in 1..n}$ *and* $\tau_c = \mathrm{close}((\forall(\alpha_i \lhd \beta_i \simeq \tau_i))^{i \in 1..n})$*. If* $\Gamma' \vdash \mathrm{ok}$*, then* $\Gamma' \vdash \tau_1 \sim \tau_c$ *and* $\Gamma \vdash \tau_c :: \star$ *holds.*

Soundness

The proof of soundness consists in subject reduction and progress. Their proof are based on the previous lemmas, and follow the same structure as the ones of Core $F^{\forall}$ (Section 2.4.5 on page 27).

Proposition 2.6.27 (Subject reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \longrightarrow M'$*, then* $\Gamma \vdash M' : \tau$*.*

The statement of *progress* had to be extended to cope with our semantics of fixpoints of terms. We did not try to prove a more refined version of progress, that uses $\notin$-reductions, as done in Section 2.4.5.

Proposition 2.6.28 (Progress)**.** *Assume* $\Gamma \vdash M : \tau$*. The following assertions hold:*

- *If* Γ *contains no value variable binding, then either* M *is reducible, or it is a result;*
- *If* M *is an extended result, then either it is reducible, or it is a result.*

2.7 Related work

As already mentioned in Section 2.1, Mitchell and Plotkin [MP88] draw a parallel between the practice of programming with abstract types, and the use of existential types. While their paper gives a type-theoretic interpretation of the notion of abstract type, the difference in the programming style that is induced by existential types is sufficiently relevant to suggest that programming with existential types is less convenient than using abstract types as known in programming languages.

Cardelli and Leroy [CL90] explore the link between existential types and a calculus equipped with the *dot notation*, that is closer to the programming practice: essentially, the witness of a value x of existential type $\exists\alpha.\tau$ is reached under the name $x.Fst$ and the opened term under $x.snd$, which has type $\tau[\alpha \leftarrow x.Fst]$. Translations from and to existential types are described: it appears that their calculus with dot notation performs implicit unpack-ings. It also looks like that the scope limitation for abstract types is still present in their calculus, because their types are not expressive enough to express dependencies.

The (revised) Definition of Standard ML [MTHM97] defines the static semantics of modules using *semantic objects*, that more or less correspond to syntactic signatures without internal dependencies. The static semantics gathers every type component: for instance, even the type components of a local module definition are kept in the resulting semantic object, thus solving the *avoidance problem*. The dynamic semantics is defined *via* an interpretation into *compound objects*, which somehow represent structures containing evaluated components, but, again, without internal dependencies.

Russo [Rus03] justifies the meaninglessness of dependent types for modules, by giving them System F types, which, by definition, are not dependent types. He first considers a stateful static semantics, that gathers the generated (abstract) type components, in the spirit of the Definition of Standard ML [MTHM97]. He then defines stateless typing rules, which track the generated type variables by keeping them as existential quantification. A proof of equivalence between the two systems is given. Dynamic semantics remains the one given in the Definition. One can think of his existential quantification, that is kept in front of types, as a kind of automatic extrusion of existential quantification, and as a means to express generativity. Note that it is done in an implicit way, while $F^{\forall}$ requires explicit manipulation of existentials. More recently, Rossberg, Russo and Dreyer [RRD10] gave a translation from a ML-like module language into System F, that is, a denotational semantics of ML in terms of the syntax of System F, and produced a formal proof of soundness, that was verified in the Coq proof assistant. Another translation is sketched in Section 4.3.2 in Chapter 4, where a detailed comparison with the former translation is given.

In the context of run-time type inspection, Rossberg [Ros03] introduces λ_N, a version of System F with a construct to define abstract types and a mechanism of directed coercions. His abstract types can be automatically extruded to allow sharper type analysis, and are thus close to our Σ binder. His coercions resemble ours, though ours are symmetric, because they never cross the abstraction barrier. Although the two systems seem kindred in spirit, they are subtly different, because they have been designed for quite different purposes: in particular, λ_N is only partially related to traditional existential types, since parametricity is purposely violated.

Dreyer [Dre07a] defines RTG, a language to handle abstract types, and that is suitable to encode recursive modules and mixins, as demonstrated in [Dre07b, DR08], with some modifications to the original RTG language. In spite of strong similarities, some *deep* technical differences remain between RTG and $F^{\forall}$. The treatment of the linear resources differs significantly: RTG's semantics employs a type store to model static but imperative type reference updates, whereas we just use extrusions of Σ binders. These two approaches might be related by seeing our extrusion as a local treatment of his type store, as has already been proposed for value references [WF94]. Dreyer uses assignment in a global store to guarantee the uniqueness of writing: this exposes the evaluation order in the typing rules of RTG and makes them asymmetrical, moving away from a logical specification, whereas we *zip* contexts to enforce sound openings and maintain a close correspondence with logic. *Intuitively,* we think of existential values as generating a fresh type when opened, while he considers them as

functions in "destination passing style" (Dps): an existential value is interpreted as a function that expects a non-initialized (*i.e.* writable) type reference, and that will internally assign it to the value of the witness; the type reference is then considered initialized, hence cannot be overwritten, and is exposed without its definition to the rest of the program to keep its definition abstract. In a nutshell, Dreyer gives an *imperative*, *stateful* interpretation to its constructs, whereas we just use notions of *scope* to achieve roughly the same behavior. Notice that his stateful interpretation naturally gives rise to a stateful dynamic semantics, that is to a semantics that threads a global type store, whereas our scope-based semantics keeps the abstract types as local as possible, and avoids the use of a global store. In more recent work [DR08], Dreyer takes a similar approach to ours for the type system, *i.e.* he bases it on linearity, that has the advantage of making the type system more independent from the dynamic semantics.

In spite of these technical differences, the two systems have similar constructs: the "new" primitive is similar to our ν binder; the "set $\alpha := \tau$ in M" is related to the $\Sigma\,\langle\alpha\rangle\,(\alpha = \tau)\,M$ construct. Note the use of a single type name here (as mentioned in Section 2.6.3 on page 40). The two systems differ a little more in other constructs. In RTG, the creation of an *impure* function of type $\tau_1 \xrightarrow{\alpha\downarrow} \tau_2$, whose body defines a witness for a type variable α, is always prefixed by the Dps construct, namely the generalization by a writable type variable $\Lambda\alpha \uparrow K.\, M$. The former is useful to write typical examples of recursive modules and allows for their separate compilation. However, this construct taken alone would have to be treated linearly, which would require the introduction of linearity in types, and would raise type wellformedness issues with respect to type substitution. Hence, the two constructs are combined into a single form. It is said that a term with type $\exists\alpha \downarrow K.\tau$ can be understood as a Dps function of type $\forall\alpha \uparrow K.() \xrightarrow{\alpha\downarrow} \tau$. In other words, an existential value is a term where the assignment for the witness is frozen. This implies, however, that the body of a Dps function, hence the body of an existential term, is *not* evaluated in RTG. One could argue that it would suffice to predefine the body with a let-binding, so that it is evaluated, but this is not always feasible since the body can depend itself on the type variable α. By contrast, $F^{\curlyvee}$ disallows the definition of impure functions, but the existential introduction $\exists\alpha.\, M$ corresponds to RTG's type variable generalization $\Lambda\alpha \uparrow K.\, M$ taken alone. However, evaluation *does* take place under existential quantifiers in $F^{\curlyvee}$. To enable this eager evaluation, we crucially rely on our *local* management of existential resources and their elimination.

In the context of dynamic linking, Abadi, Gonthier and Werner [AGW04] give a computational meaning to Hilbert's ϵ operator (the choice operator) in their System $\mathcal{E}$: the type $\epsilon\alpha.\tau$ represents a witness α for which τ holds, if such a witness exists (the type variable α is bound in τ). It is introduced with the construct $\langle e : \tau' \text{ with } \alpha = \tau\rangle$, with the following typing rule:

$$\frac{\Gamma \vdash e : \tau'[\alpha \leftarrow \tau]}{\Gamma \vdash \langle e : \tau' \text{ with } \alpha = \tau\rangle : \tau'[\alpha \leftarrow \epsilon\alpha.\tau']}$$

The ϵ operator has no elimination construct. They explain that $\tau'[\alpha \leftarrow \epsilon\alpha.\tau']$ may be viewed as the open interface type for the interface τ', and that $\exists\alpha.\tau'$ may be viewed as the closed one. The ϵ operator must be restricted to keep the type system sound: they cannot be arbitrarily nested. Then, they define a *linking* operation $e' \wr \langle e : \tau' \text{ with } \alpha = \tau\rangle$, that replaces every implementation of $\tau'[\alpha \leftarrow \epsilon\alpha.\tau']$ that occur in the term e' with the given implementation $\langle e : \tau' \text{ with } \alpha = \tau\rangle$. Even if the witnesses are different, the linking operator remains safe. The reduction relation contains call-by-value β-reduction and the following reduction rule

$$C[\langle e : \tau' \text{ with } \alpha = \tau\rangle] \longrightarrow (C \wr \langle e : \tau' \text{ with } \alpha = \tau\rangle)[e]$$

where the linking operator is extended to evaluation contexts. They authors prove that the system is type-safe, even if types are not preserved.

The introduction construct for the ϵ operator $\langle e : \tau' \text{ with } \alpha = \tau \rangle$ is similar to the term $\Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ (e : \tau')$ in $\mathsf{F}^{\curlyvee}$: they both hide occurrences of τ in the type of e. The main difference is that we use a *name* β, whereas they use the type $\epsilon\alpha.\tau'$ to refer to the type of the witness. As a consequence, the type of the witness is linked with the interface: the interface occurs in the type of the witness. Hence, the introduction construct for ϵ cannot be easily split, as what is done in $\mathsf{F}^{\curlyvee}$. We think this is the reason why types are not preserved by reduction: reduction needs to locally reveal the implementation, but this is globally done in System $\mathcal{E}$. Indeed, the linking reduction step removes the introduction construct in the current hole of the evaluation context.

Another great difference between System $\mathcal{E}$ and $\mathsf{F}^{\curlyvee}$ is the absence of any linearity constraint in System $\mathcal{E}$: the *linking* reduction step replaces every implementation of an interface in the evaluation context with the one that is currently available. This operation ensures that the uses of the interfaces remain consistent. The implementation is *dynamically* picked in System $\mathcal{E}$, whereas it is *statically* chosen in $\mathsf{F}^{\curlyvee}$: the linearity constraint is there to ensure that only one implementation is used. On the one hand, it suggests that $\mathsf{F}^{\curlyvee}$ is less expressive than a system that is equipped with the ϵ operator. On the other hand, the authors explain that the choice operator interacts with polymorphism in surprising ways.

The authors discuss the expressive power brought by the choice operator to System F: it permits to derive a proof for $\exists\alpha.((\exists\alpha.\tau) \to \tau)$, which is impossible in System F. The ϵ operator essentially brings nothing else. Interestingly, in $\mathsf{F}^{\curlyvee}$, the term $\exists\alpha.\,\lambda(x : \exists\alpha.\tau)\ \mathsf{open}\ \langle\alpha\rangle\ x$ is ill-typed because of the linearity constraint on the bodies of λ-abstractions: without this condition, it would have the type we are interested in, that is $\exists\alpha.((\exists\alpha.\tau) \to \tau)$. Our intuition is the following: in $\mathsf{F}^{\curlyvee}$ one cannot statically find a witness for the existential (which is necessary to create an existential package), because the witness will be dynamically given once the function is applied to the argument. Since System $\mathcal{E}$ relies on a sort of dynamic notion of witness, the creation of the package can be achieved.

We think that it is possible to change the semantics of $\mathsf{F}^{\curlyvee}$ to mimic the linking operation of System $\mathcal{E}$: with this modified semantics, we could remove the linearity constraint that is used in $\mathsf{F}^{\curlyvee}$'s type system. If we succeed, we would obtain a system that contains System F and in which the formula $\exists\alpha.\,\lambda(x : \exists\alpha.\tau)\ \mathsf{open}\ \langle\alpha\rangle\ x$ is derivable: a system as powerful as second-order logic equipped with choice, but without the ϵ quantifier in types. This path of research probably constitutes an interesting direction for future work.

Flatt and Felleisen [FF98b] introduced constraints within signatures to track dependencies, whereas we used constraints only in environments.

2.8 Conclusion and future work

This chapter focused on the definition for a better explicit language for handling existential types: the language $\mathsf{F}^{\curlyvee}$, whose essential ingredient is a form of *linearity*. We showed that System F can be greatly improved in this way, by defining other constructs for existential types, that allow unpacked existential packages to live in an open scope. This brings more flexibility in the definition of programs that use existential types, and we argue that is allows a programming style that is close to the style of ML modules. We equipped $\mathsf{F}^{\curlyvee}$ with a small step reduction semantics, that makes heavy use of the extrusion of one construct, in order to cope with linearity. We proved that $\mathsf{F}^{\curlyvee}$ is sound, as it enjoys the subject reduction and progress properties. A formal proof of soundness on a slight variation was developed in the Coq proof assistant. We also demonstrated that a tight connexion exists between $\mathsf{F}^{\curlyvee}$ and System F, that holds on the static and dynamic levels. Extensions that allow more programming features were also considered. Still, $\mathsf{F}^{\curlyvee}$ has some limitations, that we would like to discuss, and from which we suggest some leads to guide future work.

2.8.1 Limitations of $\mathsf{F}^{\curlyvee}$

A *purity* restriction holds for bodies of functions and of generalizations, whereas it is not the case on the constructs of RTG, for instance. We already saw in Section 2.7 that it was possible in RTG thanks to the aggregation of two distinct constructs, namely existential closure and function over an impure body. Having the latter alone would require it to be treated in a linear way, that is: one would then need *linear types* to keep the system sound. While it would certainly give the system a more regular flavor, it would also certainly make it unreasonably complex. This would have unfortunate consequences for the theoretician, for the implementer, and for the user too. Hence, this might not be a good direction to explore.

Another limitation of Core $\mathsf{F}^{\curlyvee}$ has already been foreseen in Section 2.6.4: we imposed a syntactic restriction on the bodies of fixpoints at the level of terms, so that it is possible, by reducing them, to extract every source of linearity out of the fixpoint, before unfolding it. Indeed, when reducing under value binders, such as in the bodies of fixpoints or of functions, the constructs of open existential types create new irreducible terms, that are *not pure*: for instance, the term $\mathsf{open}\ \langle\alpha\rangle\ x$ is one of these, and cannot safely be duplicated. This suggests that Core $\mathsf{F}^{\curlyvee}$ is, in some sense, incomplete, since strong reduction is not fully possible. This is however important to notice that is is already the case when one equips System F with primitive constructs for existential types: the term $\mathsf{unpack}\ x\ \mathsf{as}\ \langle\alpha, y\rangle\ \mathsf{in}\ 1$ cannot be simplified down to 1, because the unpack cannot be reduced when no pack is given to it. The situation in Core $\mathsf{F}^{\curlyvee}$ seems more obviously annoying: in the term $\mathsf{let}\ y = \mathsf{open}\ \langle\alpha\rangle\ x\ \mathsf{in}\ 1$, it seems very natural to reduce the let, so that we get 1, but it is not permitted.

We think that the two limitations we just highlighted are instances of a more general problem: we used linearity to avoid inconsistencies in witness definitions, and it lead us to introduce extrusion to respect the linearity conditions, *i.e.* to forbid some duplications or erasures. But was it our primary intention? Is not linearity just a trick not to reveal the real problem, that is *consistent sharing*? Linearity is indeed an indirect way to talk about sharing: it only copes with *non-duplication*. The above situations make us learn that there are situations where duplication is desired, but one would like to be assured at the same time, that all duplicated subterms come from the same common source: in our case, if there are two occurrences $\mathsf{open}\ \langle\alpha\rangle\ M_1$ and $\mathsf{open}\ \langle\alpha\rangle\ M_2$, then M_1 and M_2 should have a common ancestor with respect to reduction. It is clearer now that linearity was *just an easy way* to ensure this condition: there cannot be such two distinct occurrences, as this would contradict linearity.

The problem is now how to statically ensure that sharing is consistent. This is an interesting question: we are not aware of any type system that treats this topic, but we can distinguish several ways to explore it. First, considering λ-terms as dags instead of trees could do the trick, but would also raise the question of how to type term-dags, and it would also model a different reduction relation, since all the shared occurrences of a subterm should be reduced altogether in the same manner. Another lead would be to use a convertibility test in the typing rules to test for the consistency of openings, but it is not clear how to do design such a system. Finally, since sharing in the λ-calculus has been heavily studied, maybe some ideas from previous work on this topic, such as Lévy's labeling [Lé78], can be reused in a static type system.

2.8.2 Future work

Getting rid of linearity to consider properties that are more directly bound to *sharing* was described in the previous section and looks like an uncertain but challenging, long-term path of research.

Among future work remains the study of representation independence properties for Core $\mathsf{F}^{\curlyvee}$, and for the translations of Section 2.5. The differences with how this problem is treated in System F would probably highlight more particularities of $\mathsf{F}^{\curlyvee}$. The use of a proof assistant to conduct this study would probably be of valuable help.

The integration of more features, taken from real world programming languages, belongs to the

plan: an extension of Core $F^{\forall}$ with subtyping, higher order types and singleton kinds is considered in Chapter 4, and helps to write more compact programs with $F^{\forall}$. Adding value references should also be considered, and would probably raise some interesting problems, since the global store for locations of references seems to create a tension with our local treatment of abstract types: a change in the semantics would probably be required. We foresee two possibilities: either treat abstract types in a more global manner, *i.e.* extrude not only the Σs but also the νs, or, dually, handle stores for value references in a more local manner.

Extending the system with primitive sum types should not be a problem: one could already encode them using functions, but primitive sum types could be more expressive, since they would not inherit the purity restriction of functions. A possibility is to share the same environment to typecheck the branches of a case distinction, instead of zipping them: the same technique is used in linear logic, that distinguishes mutiplicative and additive connectors.

Of course, some form of type inference will eventually be needed in a surface language based on $F^{\forall}$. An easy solution is to stratify the type system, just for the purpose of type inference. We could infer ML-like types for the base level and require explicit type information for the module level, as it is currently the case in ML. Another more ambitious direction is to use a form of partial type inference with first-class polymorphism: several techniques can be envisaged (such as bidirectional type inference, implicit arguments…) and it is not clear what direction would be the more powerful and easy to use and understand. The experience gained from the implementation and the practice of implicit arguments in Coq, Agda or Scala might be pertinent.

« Tous les chemins mènent à soi. »

(Jacques Lanzmann)

Chapter 3

Type definitions and singleton kinds

A particularity of the ML module system is the existence of type components in structures and in signatures, and especially *concrete*, or *transparent* type components, that is, type components that are given a definition. The first impact of this feature is the ability to factorize type expressions using type definitions. Since type components can be exported across module boundaries, type definitions are available not only within their scopes, but can also be shared *between* modules, and even between the argument of a functor and its body.

Type components were initially modeled by attaching an equation to each concrete type definition, whereas opaque type components had no such equation. More recently, singleton kinds were used to provide a uniform treatment of type components: whether transparent or abstract, type components are uniformly specified by a kind. What distinguishes the two sorts of type components is the *accuracy* of their kinds. Singleton types and singleton kinds can indeed be used to model definitions in languages for proof assistants and to model type definitions in module systems as in ML. Singletons induce a powerful notion of equivalence. Basically, the kind $\mathcal{S}(\tau)$ represents the equivalence class of the type τ. Then, if a type τ' is proved to have a kind $\mathcal{S}(\tau)$, it means that τ' is equivalent to τ.

Systems with singletons have already been extensively studied. Aspinall [Asp95] first introduced them. Courant [Cou03] explored a way to express the equivalence relation as the reflexive symmetric transitive closure of a reduction relation: this way, he stayed close to the spirit of the Calculus of Constructions, but kept an intentional setting. Stone and Harper [SH06] defined a normalization algorithm and a procedure to check equivalence in a system with singleton kinds with extensionality. They proved them correct and complete with respect to their judgmental version of equivalence. Their proof of completeness is rather complex, and based on a logical relation. They implemented their system as a component of the TILT compiler [PCHS01]. Crary showed that the conceptual complexity induced by the singleton kinds system is in fact pretty low [Cra07]: one can eliminate singleton kinds by using η-expansion, then equivalence boils down to β-equivalence. Section 3.4 and Section 3.5 are based on this idea, and improve on it. More recently, in the purpose of the mechanization of Standard ML [LCH07], Crary used canonical forms and hereditary substitutions to prove with Twelf [HHP87] the decidability of type equivalence with singleton kinds. Indeed, Stone and Harper's proof could not be expressed in Twelf, so a workaround was needed to complete the formalization of Standard ML.

The reading of the original paper on singletons [SH06] is *strongly* recommended to better understand this chapter. The current chapter begins with a quick introduction to the Harper-Stone's system of singleton kinds (Section 3.1): this introductory reading recalls some ideas of Harper-Stone's article and is required to understand the rest of the chapter. Then, we expose in details our motivation and what we achieve in this chapter: it is purposely pushed back to Section 3.2, to make the explanation easier. Then, we illustrate our technique with a simple test case in Section 3.4: the simply typed λ-calculus with pairs, equipped with extensional equivalence. We extend our results to the system with singleton kinds in Section 3.5.

3.1 Singleton kinds: Harper-Stone system

This section contains a brief description of the singleton kind system developed by Stone and Harper. The reader is invited to read [SH06] for further details. The original system describes a language of terms that are classified by types. Since we intend to use this language as a language of types, we will deliberately describe the Stone and Harper system, by shifting it to a language of types that are described by kinds. This shift of vocabulary is already present in System F^ω: the types of F^ω are actually terms of the simply typed λ-calculus.

In the rest of the chapter, the judgments of the Harper-Stone system are indexed by the letters HS.

3.1.1 Harper-Stone's system: definitions

The syntax of kinds and types is defined as follows:

Definition 3.1.1 (Kinds and types)**.**

$$\begin{array}{lcl} \kappa & ::= & \star \mid \Pi(\alpha : \kappa)\,\kappa \mid \Sigma(\alpha : \kappa)\,\kappa \mid \mathcal{S}(\tau) \\ \tau & ::= & \alpha \mid \lambda(\alpha :: \kappa)\,\tau \mid \tau\,\tau \mid (\tau, \tau) \mid \tau.1 \mid \tau.2 \end{array}$$

Kinds are either the base kind $\star$, or dependent arrow kinds, or dependent pair kinds, or singleton kinds.

Notation. If $\alpha \notin \mathrm{ftv}(\kappa_2)$, we write $\kappa_1 \times \kappa_2$ (respectively $\kappa_1 \to \kappa_2$) instead of $\Sigma(\alpha : \kappa_1)\,\kappa_2$ (respectively $\Pi(\alpha : \kappa_1)\,\kappa_2$).

Kind wellformedness only permits us to consider singletons of types that have a base kind (rule WfKindSingle in Figure 3.1 on the next page). The types are the usual terms of a λ-calculus with pairs. The rules of the typing relation (Figure 3.3 on page 58) are the usual ones of a dependently typed λ-calculus, to which four rules have been added:

- Sub is a subkinding rule, whose judgment is described hereafter;
- Refl states that any wellformed type τ of a base kind also has the singleton kind of itself, that is the kind $\mathcal{S}(\tau)$. This rule is often called the singleton introduction rule;
- ExtPi states that any type that has an arrow kind can also be given the kind of its η-expansion;
- ExtSigma is similar to the previous rule, but for pair kinds.

The last three rules are somewhat unusual in a type system. ExtPi and ExtSigma ensure that the system is stable under η-expansions, which is false in their absence, as noticed by Stone and Harper. We think they are necessary because of the restriction of singletons at the base kind. Extending them to higher kinds is done in their paper, and explained in Definition 3.1.2 on page 60.

The subkinding judgment is defined in Figure 3.2 on the facing page. It is, as usual, contravariant on the domains of functions, and covariant on their co-domains, and on components of pairs. It is otherwise generated by three rules:

- SubStar ensures that the base kind is a subkind of itself;
- SubForget allows one to forget a definition: any wellformed singleton kind is a subkind of the base kind;
- SubSingle allows to replace a type inside a singleton kind with an equivalent one.

$$\textsc{WfEnvEmpty}\ \frac{}{\varepsilon \vdash_{\mathsf{HS}} \mathrm{ok}} \qquad \textsc{WfEnvCons}\ \frac{\Gamma \vdash_{\mathsf{HS}} \kappa\ \mathrm{ok} \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \alpha :: \kappa \vdash_{\mathsf{HS}} \mathrm{ok}}$$

$$\textsc{WfKindStar}\ \frac{\Gamma \vdash_{\mathsf{HS}} \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \star\ \mathrm{ok}} \qquad \textsc{WfKindSingle}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \star}{\Gamma \vdash_{\mathsf{HS}} \mathcal{S}(\tau)\ \mathrm{ok}} \qquad \textsc{WfKindPi}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2\ \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\,\kappa_2\ \mathrm{ok}} \qquad \textsc{WfKindSigma}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2\ \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\,\kappa_2\ \mathrm{ok}}$$

Figure 3.1: Wellformed environments and wellformed kinds.

$$\textsc{SubForget}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \star}{\Gamma \vdash_{\mathsf{HS}} \mathcal{S}(\tau) \leqslant \star} \qquad \textsc{SubSingle}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash_{\mathsf{HS}} \mathcal{S}(\tau_1) \leqslant \mathcal{S}(\tau_2)} \qquad \textsc{SubStar}\ \frac{\Gamma \vdash_{\mathsf{HS}} \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \star \leqslant \star}$$

$$\textsc{SubPi}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\,\kappa_2\ \mathrm{ok} \qquad \Gamma \vdash_{\mathsf{HS}} \kappa_1' \leqslant \kappa_1 \qquad \Gamma, \alpha :: \kappa_1' \vdash_{\mathsf{HS}} \kappa_2 \leqslant \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\,\kappa_2 \leqslant \Pi(\alpha : \kappa_1')\,\kappa_2'}$$

$$\textsc{SubSigma}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1')\,\kappa_2'\ \mathrm{ok} \qquad \Gamma \vdash_{\mathsf{HS}} \kappa_1 \leqslant \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2 \leqslant \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\,\kappa_2 \leqslant \Sigma(\alpha : \kappa_1')\,\kappa_2'}$$

$$\textsc{EqStar}\ \frac{\Gamma \vdash_{\mathsf{HS}} \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \star \equiv \star} \qquad \textsc{EqSingle}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash_{\mathsf{HS}} \mathcal{S}(\tau_1) \equiv \mathcal{S}(\tau_2)}$$

$$\textsc{EqPi}\ \frac{\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2 \equiv \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\,\kappa_2 \equiv \Pi(\alpha : \kappa_1')\,\kappa_2'} \qquad \textsc{EqSigma}\ \frac{\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2 \equiv \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\,\kappa_2 \equiv \Sigma(\alpha : \kappa_1')\,\kappa_2'}$$

Figure 3.2: Subkinding and kind equivalence.

$$\textsc{Var}\ \frac{\Gamma \vdash_{\mathsf{HS}} \mathrm{ok} \qquad \alpha :: \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{HS}} \alpha :: \kappa} \qquad \textsc{Lam}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau :: \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \lambda(\alpha :: \kappa_1)\,\tau :: \Pi(\alpha : \kappa_1)\,\kappa_2} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 :: \Pi(\alpha : \kappa_2)\,\kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 :: \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau_1\ \tau_2 :: \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\,\kappa_2\ \mathrm{ok} \qquad \Gamma \vdash_{\mathsf{HS}} \tau_1 :: \kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 :: \kappa_2[\alpha \leftarrow \tau_1]}{\Gamma \vdash_{\mathsf{HS}} (\tau_1, \tau_2) :: \Sigma(\alpha : \kappa_1)\,\kappa_2} \qquad \textsc{ProjL}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau.1 :: \kappa_1} \qquad \textsc{ProjR}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau.2 :: \kappa_2[\alpha \leftarrow \tau.1]}$$

$$\textsc{Refl}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \star}{\Gamma \vdash_{\mathsf{HS}} \tau :: \mathcal{S}(\tau)} \qquad \textsc{ExtPi}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau\ \alpha :: \kappa_2 \quad \Gamma \vdash_{\mathsf{HS}} \tau :: \Pi(\alpha : \kappa_1)\,\kappa_2' \qquad \Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\,\kappa_2'\ \mathrm{ok}}{\Gamma \vdash_{\mathsf{HS}} \tau :: \Pi(\alpha : \kappa_1)\,\kappa_2}$$

$$\textsc{ExtSigma}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\,\kappa_2\ \mathrm{ok} \quad \Gamma \vdash_{\mathsf{HS}} \tau.1 :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \tau.2 :: \kappa_2[\alpha \leftarrow \tau.1]}{\Gamma \vdash_{\mathsf{HS}} \tau :: \Sigma(\alpha : \kappa_1)\,\kappa_2} \qquad \textsc{Sub}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \kappa_1 \leqslant \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa_2}$$

Figure 3.3: Wellformed types.

The kind equivalence judgment is identical to subkinding, from which the SubForget rule has been removed, so as to make it symmetric.

The main judgment is the type equivalence judgment (Figure 3.4 on the next page). The majority of the rules are standard rules to close the judgment under reflexivity, symmetry, transitivity and congruence. We describe the remaining ones:

- EqExtSingle specifies that having a singleton kind $\mathcal{S}(\tau_2)$ means being equivalent to τ_2. This rule is often called the singleton elimination rule;
- EqExtPi closes the equivalence under extensionality at arrow kinds;
- EqExtSigma closes the equivalence under extensionality at pair kinds;
- EqSub closes the equivalence relation under subkinding.

3.1.2 Examples

It is not obvious, at first glance, what these definitions allow or not. We recall some examples taken from [SH06], or inspired from the same article. Under the hypothesis that β has kind $\mathcal{S}(\alpha)$ (*i.e.* that β is equivalent to α), the pair (α, β) is equivalent to its *flipped* version:

$$\alpha : \star, \beta : \mathcal{S}(\alpha) \vdash (\alpha, \beta) \equiv (\beta, \alpha) :: \star \times \star$$

The next examples are about *partial* definitions, that is, functions or pairs, that are partially specified to be equivalent to something else. For instance, under the hypothesis that f is a function that always returns the value α, then f is equivalent to a constant function that returns α:

$$\alpha : \star, f : \Pi(\beta : \star)\,\mathcal{S}(\alpha) \vdash f \equiv \lambda(\beta :: \star)\,\alpha :: \star \to \star$$

$$\textsc{EqRefl}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa}{\Gamma \vdash_{\mathsf{HS}} \tau \equiv \tau :: \kappa} \qquad \textsc{EqSym}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv \tau_1 :: \kappa}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa} \qquad \textsc{EqTrans}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa \qquad \Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv \tau_3 :: \kappa}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_3 :: \kappa}$$

$$\textsc{EqLam}\ \frac{\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_2 \qquad \Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa'}{\Gamma \vdash_{\mathsf{HS}} \lambda(\alpha :: \kappa_1)\, \tau_1 \equiv \lambda(\alpha :: \kappa_2)\, \tau_2 :: \Pi(\alpha : \kappa_1)\, \kappa'} \qquad \textsc{EqApp}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_1' :: \Pi(\alpha : \kappa_2)\, \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv \tau_2' :: \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' :: \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{EqProjL}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \Sigma(\alpha : \kappa_1)\, \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau_1.1 \equiv \tau_2.1 :: \kappa_1} \qquad \textsc{EqProjR}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \Sigma(\alpha : \kappa_1)\, \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau_1.2 \equiv \tau_2.2 :: \kappa_2[\alpha \leftarrow \tau_1.1]}$$

$$\textsc{EqPair}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\, \kappa_2\ \mathrm{ok} \qquad \Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_1' :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv \tau_2' :: \kappa_2[\alpha \leftarrow \tau_1]}{\Gamma \vdash_{\mathsf{HS}} (\tau_1, \tau_2) \equiv (\tau_1', \tau_2') :: \Sigma(\alpha : \kappa_1)\, \kappa_2} \qquad \textsc{EqExtPi}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau_1\ \alpha \equiv \tau_2\ \alpha :: \kappa_2 \qquad \Gamma \vdash_{\mathsf{HS}} \tau_1 :: \Pi(\alpha : \kappa_1)\, \kappa_3 \qquad \Gamma \vdash_{\mathsf{HS}} \tau_2 :: \Pi(\alpha : \kappa_1)\, \kappa_4}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \Pi(\alpha : \kappa_1)\, \kappa_2}$$

$$\textsc{EqExtSigma}\ \frac{\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1)\, \kappa_2\ \mathrm{ok} \qquad \Gamma \vdash_{\mathsf{HS}} \tau_1.1 \equiv \tau_2.1 :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \tau_1.2 \equiv \tau_2.2 :: \kappa_2[\alpha \leftarrow \tau_1.1]}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \Sigma(\alpha : \kappa_1)\, \kappa_2} \qquad \textsc{EqExtSingle}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 :: \mathcal{S}(\tau_2)}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \mathcal{S}(\tau_2)}$$

$$\textsc{EqSub}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{HS}} \kappa_1 \leqslant \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa_2}$$

Figure 3.4: Type equivalence.

Under the hypothesis that f is a function that expects α as argument, f is equivalent to a function that expects α and returns the result of f:

$$\alpha : \star, f : \mathcal{S}(\alpha) \to \star \vdash f \equiv \lambda(\beta :: \mathcal{S}(\alpha))\, f\ \alpha :: \star \to \star$$

Under the assumption that y is a pair whose second component is equal to x, then the second projection of y is indeed equal to x:

$$\alpha : \star, \beta : \star \times \mathcal{S}(\alpha) \vdash \beta.2 \equiv \alpha :: \star$$

Under the hypothesis that α is a pair whose second component is equal to the first one, then one can prove that its two projections are equivalent, but also that α is equivalent to the pair that is composed of twice the first projection of α:

$$\alpha : \Sigma(\beta : \star)\, \mathcal{S}(\beta) \vdash \alpha.1 \equiv \alpha.2 :: \star$$
$$\alpha : \Sigma(\beta : \star)\, \mathcal{S}(\beta) \vdash \alpha \equiv (\alpha.1, \alpha.1) :: \star \times \star$$

It is rather obvious that the equivalence judgment is sensitive to the context, since it contains the hypotheses about equivalence that are available for reasoning. It is however less clear that equivalence is also sensitive to the kind at which it is considered. For instance, one cannot prove that the identity is equivalent to a constant function, and the following does not hold:

$$\alpha : \star \vdash \lambda(\beta :: \star)\, \beta \equiv \lambda(\beta :: \star)\, \alpha :: \star \to \star$$

However, if the same judgment is considered at a kind that forces the functions to expect α as argument, then the same two functions become equivalent:

$$\alpha : \star \vdash \lambda(\beta :: \star)\, \beta \equiv \lambda(\beta :: \star)\, \alpha :: \mathcal{S}(\alpha) \to \star$$

One can parametrize the same example to show how embedded singleton kinds impact the judgment:

$$\alpha : \star, f : (\mathcal{S}(\alpha) \to \star) \to \star \vdash f\ \lambda(\beta :: \star)\, \beta \equiv f\ \lambda(\beta :: \star)\, \alpha :: \star$$

holds, while

$$\alpha : \star, f : (\star \to \star) \to \star \vdash f\ \lambda(\beta :: \star)\, \beta \equiv f\ \lambda(\beta :: \star)\, \alpha :: \star$$

does not hold.

A difficulty with the Harper-Stone system is that all judgments are mutually defined, which makes meta-theoretic proofs difficult. As we showed above, another challenge is that type equivalence is sensitive to the context and to the kind at which it is consiedered: this makes equivalence a subtle relation that is hard to decide.

3.1.3 Harper-Stone's system: properties

The rule WfKindSingle only accepts to build singletons of types that have the base kind. It is possible to define singletons at higher kinds, also called labeled singletons: the kind $\mathcal{S}_\kappa(\tau)$ is the singleton of the type τ at kind κ. Labeled singletons are defined as follows:

Definition 3.1.2 (Labeled singletons)**.**

$$\begin{array}{rcl}
\mathcal{S}_\star(\tau) & \triangleq & \mathcal{S}(\tau) \\
\mathcal{S}_{\mathcal{S}(\tau')}(\tau) & \triangleq & \mathcal{S}(\tau') \\
\mathcal{S}_{\Pi(\alpha:\kappa_1)\,\kappa_2}(\tau) & \triangleq & \Pi(\alpha : \kappa_1)\, \mathcal{S}_{\kappa_2}(\tau\ \alpha) \\
\mathcal{S}_{\Sigma(\alpha:\kappa_1)\,\kappa_2}(\tau) & \triangleq & \mathcal{S}_{\kappa_1}(\tau.1) \times \mathcal{S}_{\kappa_2[\alpha \leftarrow \tau.1]}(\tau.2)
\end{array}$$

The definition follows a scheme that is close to extensionality, but at the level of kinds. It is important to notice that labeled singletons do not always make sense: for instance, $\mathcal{S}_{\Sigma(\beta:\kappa_1)\,\kappa_2}(\lambda(\alpha :: \star)\,\alpha)$ is not a wellformed kind, because $\lambda(\alpha :: \star)\,\alpha$ cannot have the kind $\Sigma(\beta : \kappa_1)\,\kappa_2$. Similarly, $\mathcal{S}_{\mathcal{S}(\tau_1)}(\tau_2)$ only makes sense when τ_1 and τ_2 are two equivalent types. As a consequence, a labeled singleton kind $\mathcal{S}_\kappa(\tau)$ will be built only when τ has the kind κ.

We now review essential properties of the Harper-Stone singleton system. We do not present them in an order that is compatible with the dependencies of their proofs. Complete proofs can be found in [SH06].

The two propositions deal with validity of the relations, that is to say: every wellformed judgment involves wellformed arguments.

Proposition 3.1.1 (Environment validity). *If* $\Gamma \vdash \kappa\ \mathrm{ok}$ *or* $\Gamma \vdash \tau :: \kappa$ *or* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ *or* $\Gamma \vdash \kappa_1 \equiv \kappa_2$ *or* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$ *holds, then there is a subderivation with conclusion* $\Gamma \vdash \mathrm{ok}$.

Proposition 3.1.2 (Validity). *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \kappa$, *then* $\Gamma \vdash \kappa\ \mathrm{ok}$;
- *If* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ *or* $\Gamma \vdash \kappa_1 \equiv \kappa_2$, *then* $\Gamma \vdash \kappa_1\ \mathrm{ok}$ *and* $\Gamma \vdash \kappa_2\ \mathrm{ok}$;
- *If* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$, *then* $\Gamma \vdash \kappa\ \mathrm{ok}$ *and* $\Gamma \vdash \tau_1 :: \kappa$ *and* $\Gamma \vdash \tau_2 :: \kappa$.

The next propositions deal with properties on subkinding and kind equivalence: they state that subkinding is a pre-order and kind equivalence an equivalence relation, and that kind equivalence constitutes the diagonal of subkinding.

Proposition 3.1.3 (Reflexivity of subkinding and kind equivalence). *If* $\Gamma \vdash \kappa\ \mathrm{ok}$, *then* $\Gamma \vdash \kappa \leqslant \kappa$ *and* $\Gamma \vdash \kappa \equiv \kappa$.

Proposition 3.1.4 (Antisymmetry of subkinding). *If* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ *and* $\Gamma \vdash \kappa_2 \leqslant \kappa_1$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_2$.

Proposition 3.1.5 (Symmetry and transitivity of kind equivalence). *The following assertions hold:*

- *If* $\Gamma \vdash \kappa_1 \equiv \kappa_2$, *then* $\Gamma \vdash \kappa_2 \equiv \kappa_1$;
- *If* $\Gamma \vdash \kappa_1 \equiv \kappa_2$ *and* $\Gamma \vdash \kappa_2 \equiv \kappa_3$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_3$.

Proposition 3.1.6 (Inclusion of kind equivalence in subkinding). *If* $\Gamma \vdash \kappa_1 \equiv \kappa_2$, *then* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$.

Proposition 3.1.7 (Transitivity of subkinding). *If* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ *and* $\Gamma \vdash \kappa_2 \leqslant \kappa_3$, *then* $\Gamma \vdash \kappa_1 \leqslant \kappa_3$.

The following proposition gathers some admissible rules.

Proposition 3.1.8 (Admissible rules). *The following rules are admissible:*

$$\textsc{Forget}'\ \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \mathcal{S}_\kappa(\tau) \leqslant \kappa} \qquad \textsc{SubSingle}'\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa_1 \qquad \Gamma \vdash \kappa_1 \leqslant \kappa_2}{\Gamma \vdash \mathcal{S}_{\kappa_1}(\tau_1) \leqslant \mathcal{S}_{\kappa_2}(\tau_2)} \qquad \textsc{Refl}'\ \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau :: \mathcal{S}_\kappa(\tau)}$$

$$\textsc{Beta}\ \frac{\Gamma, \alpha :: \kappa_2 \vdash \tau_1 :: \kappa_1 \qquad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash (\lambda(\alpha :: \kappa_2)\,\tau_1)\ \tau_2 \equiv \tau_1[\alpha \leftarrow \tau_2] :: \kappa_1[\alpha \leftarrow \tau_2]} \qquad \textsc{Pi}\ \frac{\Gamma \vdash \tau_1 :: \kappa_1 \qquad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash (\tau_1, \tau_2).\mathrm{i} \equiv \tau_\mathrm{i} :: \kappa_\mathrm{i}}$$

$$\textsc{EtaPi}\ \frac{\Gamma \vdash \tau :: \Pi(\alpha : \kappa_1)\,\kappa_2}{\Gamma \vdash \tau \equiv \lambda(\alpha :: \kappa_1)\,\tau\ \alpha :: \Pi(\alpha : \kappa_1)\,\kappa_2} \qquad \textsc{EtaSigma}\ \frac{\Gamma \vdash \tau :: \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \vdash \tau \equiv (\tau.1, \tau.2) :: \Sigma(\alpha : \kappa_1)\,\kappa_2}$$

$$\textsc{NatKindVar}\ \frac{}{\Gamma \triangleright \alpha \uparrow \Gamma(\alpha)} \qquad \textsc{NatKindProjL}\ \frac{\Gamma \triangleright p \uparrow \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \triangleright p.1 \uparrow \kappa_1} \qquad \textsc{NatKindProjR}\ \frac{\Gamma \triangleright p \uparrow \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \triangleright p.2 \uparrow \kappa_2[\alpha \leftarrow p.1]} \qquad \textsc{NatKindApp}\ \frac{\Gamma \triangleright p \uparrow \Pi(\alpha : \kappa_1)\,\kappa_2}{\Gamma \triangleright p\ \tau \uparrow \kappa_2[\alpha \leftarrow \tau]}$$

Figure 3.5: Natural kind.

Rule Forget′ generalizes rule Forget to labeled singletons, whereas rule SubSingle′ generalizes rule SubSingle (*cf.* Figure 3.2 on page 57). The rule Refl′ extends rule Refl to higher kinds as well (*cf.* Figure 3.3 on page 58). The rules Beta and Pi state the admissibility of β-equivalence for functions and pairs; the rules ExtPi and ExtSigma cope with η-equivalence for functions and pairs.

The next proposition is the usual substitution lemma on judgments.

Proposition 3.1.9. *Assume* $\Gamma \vdash \tau :: \kappa$. *The following assertions hold:*

- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \mathrm{ok}$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \mathrm{ok}$;
- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \kappa'\ \mathrm{ok}$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \kappa'[\alpha \leftarrow \tau]\ \mathrm{ok}$;
- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \kappa_1 \equiv \kappa_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \kappa_1[\alpha \leftarrow \tau] \equiv \kappa_2[\alpha \leftarrow \tau]$;
- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \kappa_1 \leqslant \kappa_2$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \kappa_1[\alpha \leftarrow \tau] \leqslant \kappa_2[\alpha \leftarrow \tau]$;
- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \tau' :: \kappa'$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \tau'[\alpha \leftarrow \tau] :: \kappa'[\alpha \leftarrow \tau]$;
- *If* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \tau_1 \equiv \tau_2 :: \kappa'$, *then* $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash \tau_1[\alpha \leftarrow \tau] \equiv \tau_2[\alpha \leftarrow \tau] :: \kappa'[\alpha \leftarrow \tau]$;

The next proposition states that singletons can indeed be interpreted as definitions: type equivalence and kind equivalence contains the unfolding of singleton kinds.

Proposition 3.1.10. *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \kappa$ *and* $\Gamma, \alpha :: \kappa \vdash \tau' :: \kappa'$, *then* $\Gamma, \alpha :: \mathcal{S}_\kappa(\tau) \vdash \tau' \equiv \tau'[\alpha \leftarrow \tau] :: \kappa'$;
- *If* $\Gamma \vdash \tau :: \kappa$ *and* $\Gamma, \alpha :: \kappa \vdash \kappa'\ \mathrm{ok}$, *then* $\Gamma, \alpha :: \mathcal{S}_\kappa(\tau) \vdash \kappa' \equiv \kappa'[\alpha \leftarrow \tau]$.

Corollary 3.1.11. *If* $\Gamma \vdash \tau :: \kappa$ *and* $\Gamma, \alpha :: \kappa \vdash \kappa'\ \mathrm{ok}$, *then* $\Gamma \vdash \Pi(\alpha : \mathcal{S}_\kappa(\tau))\,\kappa' \equiv \Pi(\alpha : \mathcal{S}_\kappa(\tau))\,\kappa'[\alpha \leftarrow \tau]$ *and* $\Gamma \vdash \Sigma(\alpha : \mathcal{S}_\kappa(\tau))\,\kappa' \equiv \Sigma(\alpha : \mathcal{S}_\kappa(\tau))\,\kappa'[\alpha \leftarrow \tau]$.

3.1.4 Harper-Stone normalization algorithm and decidability result

Stone and Harper [SH06] give an algorithm to decide the wellformedness of types, that is based on the minimal kind property, and on an algorithm to decide type equivalence. The key component of the latter is a normalization procedure, that is proved sound and complete with respect to the specification of the system. Although the soundness property is rather easy to prove, the completeness result requires an involved proof, that is based on a logical relation. In Section 3.5, we reuse the logical relation-based technique to prove a completeness property. In the current section, we briefly recall the normalization algorithm.

One can sum it up as follows: the algorithm consists in two stages. First, the algorithm is kind-directed, and performs head η-expansions. This is done by the type normalization judgment in Figure 3.8 on the facing page. Once a base kind has been reached, types are reduced (Figure 3.6 and Figure 3.7 on the next page) following the normal strategy (*i.e.* the leftmost innermost strategy) by performing β-reduction on function applications and projected pairs, as well as the *unfolding of definitions*: this is done by replacing a type τ with another type τ' when τ has the *natural kind* $\mathcal{S}(\tau')$.

$$\textsc{HdRedBeta}\ \frac{}{\Gamma \rhd \mathcal{E}[(\lambda(\alpha :: \kappa)\,\tau)\ u] \rightsquigarrow \mathcal{E}[\tau[\alpha \leftarrow u]]} \qquad \textsc{HdRedPi}\ \frac{i \in \{1,2\}}{\Gamma \rhd \mathcal{E}[(\tau_1,\tau_2).i] \rightsquigarrow \mathcal{E}[\tau_i]} \qquad \textsc{HdRedUnfold}\ \frac{\Gamma \rhd p \uparrow \mathcal{S}(u)}{\Gamma \rhd \mathcal{E}[p] \rightsquigarrow \mathcal{E}[u]}$$

Figure 3.6: Head reduction.

$$\textsc{HdNormTrans}\ \frac{\Gamma \rhd \tau \rightsquigarrow \tau' \qquad \Gamma \rhd \tau' \Downarrow \tau''}{\Gamma \rhd \tau \Downarrow \tau''} \qquad \textsc{HdNormRefl}\ \frac{\text{there is no } \tau' \text{ such that } \Gamma \rhd \tau \rightsquigarrow \tau'}{\Gamma \rhd \tau \Downarrow \tau}$$

Figure 3.7: Head normalization.

$$\textsc{TypeNormStar}\ \frac{\Gamma \rhd \tau \Downarrow \tau' \qquad \Gamma \rhd \tau' \longrightarrow \tau'' \uparrow \star}{\Gamma \rhd \tau :: \star \Longrightarrow \tau''} \qquad \textsc{TypeNormSingle}\ \frac{\Gamma \rhd \tau \Downarrow \tau' \qquad \Gamma \rhd \tau' \longrightarrow \tau'' \uparrow \star}{\Gamma \rhd \tau :: \mathcal{S}(\tau_0) \Longrightarrow \tau''}$$

$$\textsc{TypeNormPi}\ \frac{\Gamma \rhd \kappa_1 \Longrightarrow \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \rhd \tau\ \alpha :: \kappa_2 \Longrightarrow \tau'}{\Gamma \rhd \tau :: \Pi(\alpha : \kappa_1)\,\kappa_2 \Longrightarrow \lambda(\alpha :: \kappa_1')\,\tau'} \qquad \textsc{TypeNormSigma}\ \frac{\Gamma \rhd \tau.1 :: \kappa_1 \Longrightarrow \tau_1' \qquad \Gamma \rhd \tau.2 :: \kappa_2[\alpha \leftarrow \tau.1] \Longrightarrow \tau_2'}{\Gamma \rhd \tau :: \Sigma(\alpha : \kappa_1)\,\kappa_2 \Longrightarrow (\tau_1', \tau_2')}$$

Figure 3.8: Type normalization.

$$\textsc{PathNormVar}\ \frac{}{\Gamma \rhd \alpha \longrightarrow \alpha \uparrow \Gamma(\alpha)} \qquad \textsc{PathNormApp}\ \frac{\Gamma \rhd p \longrightarrow p' \uparrow \Pi(\alpha : \kappa_1)\,\kappa_2 \qquad \Gamma \rhd \tau :: \kappa_1 \Longrightarrow \tau'}{\Gamma \rhd p\ \tau \longrightarrow p'\ \tau' \uparrow \kappa_2[\alpha \leftarrow \tau]}$$

$$\textsc{PathNormProjL}\ \frac{\Gamma \rhd p \longrightarrow p' \uparrow \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \rhd p.1 \longrightarrow p'.1 \uparrow \kappa_1} \qquad \textsc{PathNormProjR}\ \frac{\Gamma \rhd p \longrightarrow p' \uparrow \Sigma(\alpha : \kappa_1)\,\kappa_2}{\Gamma \rhd p.2 \longrightarrow p'.2 \uparrow \kappa_2[\alpha \leftarrow p.1]}$$

Figure 3.9: Path normalization.

$$\textsc{KindNormStar}\ \frac{}{\Gamma \rhd \star \Longrightarrow \star} \qquad \textsc{KindNormSingle}\ \frac{\Gamma \rhd \tau :: \star \Longrightarrow \tau'}{\Gamma \rhd \mathcal{S}(\tau) \Longrightarrow \mathcal{S}(\tau')}$$

$$\textsc{KindNormPi}\ \frac{\Gamma \rhd \kappa_1 \Longrightarrow \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \rhd \kappa_2 \Longrightarrow \kappa_2'}{\Gamma \rhd \Pi(\alpha : \kappa_1)\,\kappa_2 \Longrightarrow \Pi(\alpha : \kappa_1')\,\kappa_2'} \qquad \textsc{KindNormSigma}\ \frac{\Gamma \rhd \kappa_1 \Longrightarrow \kappa_1' \qquad \Gamma, \alpha :: \kappa_1 \rhd \kappa_2 \Longrightarrow \kappa_2'}{\Gamma \rhd \Sigma(\alpha : \kappa_1)\,\kappa_2 \Longrightarrow \Sigma(\alpha : \kappa_1')\,\kappa_2'}$$

Figure 3.10: Kind normalization.

Natural kinds are defined in Figure 3.5 on page 62, and consists of kinding a path p without making use of the Refl, ExtPi, ExtSigma or of the Sub rule. A path p is defined as follows as the closure of type variables under application and projection.

$$\mathcal{E} ::= [\cdot] \quad | \quad \mathcal{E}\ \tau \quad | \quad \mathcal{E}.i \qquad\qquad p ::= \mathcal{E}[\alpha]$$

Note that abstractions and pairs are *not* included in paths.

The singleton kinds system enjoys the minimal kind property, *i.e.* for any wellformed type, there exists a kind for this type that is lesser than any other one. Minimal kinds are discussed in Section 3.5.9 and defined in Figure 3.12 on page 99.

Natural kinds are very close to *minimal kinds*: Harper and Stone show that if a wellformed type τ has the *natural* kind κ, then its *minimal* kind is equivalent to $\mathcal{S}_\kappa(\tau)$. Once types are reduced to *paths*, then one proceeds with path normalization (Figure 3.9 on the preceding page), that normalizes the types of the right parts of path applications. Since normalization of types is kind directed, path normalization maintains the natural kind of the current path while it is being normalized.

The normalization procedure of types is recursively defined with the normalization of kinds (Figure 3.10 on the previous page), that only consists in a closure of type normalization under contexts of kinds: normalization of kinds lifts type normalization within singletons to kinds.

Stone and Harper prove that his algorithm is sound and complete with respect to type equivalence:

Theorem 3.1.12 (Stone-Harper: adequacy of normalization)**.** *Assume* $\Gamma \vdash \tau_1 :: \kappa$ *and* $\Gamma \vdash \tau_2 :: \kappa$*. Then* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$ *holds iff there exists* τ' *such that* $\Gamma \triangleright \tau_1 :: \kappa \Longrightarrow \tau'$ *and* $\Gamma \triangleright \tau_2 :: \kappa \Longrightarrow \tau'$ *holds.*

They also define a more efficient algorithm that decides type equivalence without fully normalizing the types that are tested for equivalence.

3.2 Goals of this chapter

In this chapter, we define an alternate way to test types for equivalence, by defining a reduction relation that is non-deterministic, confluent and strongly normalizing, and that combines η-expansion and β-reduction. We prove it sound and complete with respect to the Stone-Harper definition.

Our technique inherits from Crary's elimination of singleton kinds[Cra07]: he interprets the unfolding of definitions as the η-expansion of a type at the singleton kind. Expansions at other kinds permit to access deep definitions, such as singletons nested in pairs or in arguments of functions. This way, by η-expanding the free type variables of types and performing a head η-expansion, he shows that it suffices then to test types for β-convertibility only. This is to put together with Goguen [Gog05b, Gog05a], who also shows, for different type theories, that β-convertibility is enough to compare for η-convertibility, provided enough η-expansions have been performed.

The originality of our technique relies in combining β-reduction and η-expansion, while maintaining strong normalization and keeping usual evaluation contexts. When studying systems with $\beta\eta$-equivalence, Kesner and Di Cosmo [CK93], managed to define a strongly normalizing reduction for simply typed λ-calculus with pairs, sums, recursion and terminal object, but they had to restrict the evaluation contexts to succeed, so that their reduction relation is no longer defined as the closure under congruence of head reduction. By contrast, we proceed by inserting explicits marks of expansion, called *expansors*, at every occurrence of variables; then expansion of the expansors, which mimics usual kind-directed η-expansion, can be interleaved with β-reduction, and is allowed in any context. We prove that the resulting notion of convertibility coincides with Stone-Harper's definition.

Our characterization of type equivalence generalizes both Crary's method for deciding type equivalence by singletons erasure, and the normalization algorithm of [SH06]. Interestingly, our approach permits to interleave β-reduction with η-expansions, and is not restricted to follow the

normal order: since it is based on a simple, confluent calculus, more reduction strategies can be considered, and existing techniques for efficient reduction of λ-terms can possibly be reused. A difference remains: we add expansors at every occurrence of variables, not only at free occurrences.

The rest of the chapter is organized as follows: Section 3.3 introduces preliminary combination properties on confluence and normalization. Section 3.4 expounds our method of explicit expansors on the easier subcase of the simply typed λ-calculus. It is then extended to singleton kinds in Section 3.5, before the exposition of our concluding remarks in Section 3.7 on page 105.

3.3 Preliminary results: some composition properties for rewriting systems

In this section, we recall results on combination of confluence and normalization for binary relations, that are used in Section 3.4 and Section 3.5. We verified these intermediate results in the Coq system. The proof script is available at `http://gallium.inria.fr/~montagu/proofs/Rel/`. None of the results that are presented in this section are new.

Definition 3.3.1 (Terminating element)**.** Let $\mathcal{R}$ be a binary relation over a set E. Then an element $x \in E$ is said *terminating* with $\mathcal{R}$ if for any y such that $x \mathcal{R} y$ holds, then y is terminating with $\mathcal{R}$.

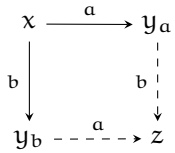
Definition 3.3.2 (Termination)**.** A binary relation $\mathcal{R}$ on a set E *terminates* if every element $x \in E$ is terminating with $\mathcal{R}$. In this case, we also say that $\mathcal{R}$ is terminating, or strongly normalizing.

Definition 3.3.3 (Normal form)**.** We say that x is a normal form for $\mathcal{R}$ (or a $\mathcal{R}$-normal form) if there is no y such that $x \mathcal{R} y$. If $z \mathcal{R} x$ and x is a $\mathcal{R}$-normal form, we may use the notation $z \mathcal{R}{\downarrow} x$.

Definition 3.3.4 (Preservation of normal forms)**.** We say that $\mathcal{R}_1$ preserves $\mathcal{R}_2$-normal forms when for every x that is a $\mathcal{R}_2$-normal form, and every y such that $x \mathcal{R}_1 y$, then y is a $\mathcal{R}_2$-normal form.

The notion of terminating element with a relation (respectively, of a terminating relation) is deliberately chosen to be the flipped version of the well known definition of *accessibility* of an element through a relation (respectively, of a *well founded* relation). We choose to use the flipped versions, because they directly apply to reduction relations, which are oriented from left to right. In the rest of this section, we will use arrows to represent binary relations over a fixed set E. Let $\to^+$ denote the transitive closure of the relation $\to$, and $\to^\star$ denote its reflexive transitive closure, and $\to^?$ denote its reflexive closure. Let $\leftrightarrow$ denote the reflexive symmetric transitive closure of the relation $\to$. Let $\to_a \cup \to_b$ denote the union of the relations $\to_a$ and $\to_b$.

Definition 3.3.5 (Commutation of relations)**.** Two relations $\to_a$ and $\to_b$ commute if the following diagram holds:



Or, formally, for every x, y_a and y_b, if $x \to_a y_a$ and $x \to_b y_b$, then there exists z such that $y_a \to_b z$ and $y_b \to_a z$.

Lemma 3.3.1. *If $\to_a$ commutes with $\to_b$, then $\to_a^\star$ commutes with $\to_b^\star$.*

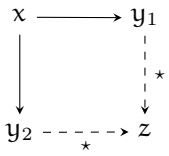
Definition 3.3.6 (Diamond)**.** A relation has the diamond property if it commutes with itself.

Definition 3.3.7 (Confluence)**.** A relation $\to$ is said *confluent* if $\to^\star$ has the diamond property.

Lemma 3.3.2. *A confluent relation has unique normal forms.*

Lemma 3.3.3. *If a relation has the diamond property, then it is confluent.*

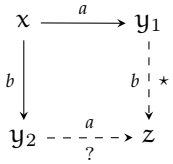
Definition 3.3.8 (Weak confluence)**.** A relation is said *weakly confluent* if the following diagram holds:



Or, formally, for every x, y_1 and y_2, if $x \to y_1$ and $x \to y_2$, then there exists z such that $y_1 \to^\star z$ and $y_2 \to^\star z$.

The following lemma is often used to prove the commutation of two relations.

Lemma 3.3.4 (Commutation condition)**.** *If the following diagram holds*

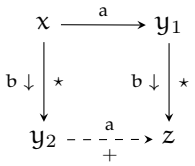


then $\to_a^\star$ commutes with $\to_b^\star$.

Lemma 3.3.5 (Newman)**.** *If a terminating relation is weakly confluent, then it is confluent.*

Lemma 3.3.6 (Hindley-Rosen [Bar84])**.** *Assume that $\to_a$ and $\to_b$ have the diamond property. Then, if $\to_a$ commutes with $\to_b$, then $\to_a \cup \to_b$ is confluent.*

Definition 3.3.9 (Commutation with normal forms)**.** We say that $\to_a$ commutes with $\to_b$-normal forms when the following diagram holds:

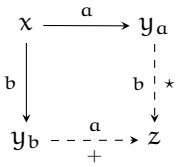


Lemma 3.3.7 (Hindley-Rosen, variant)**.** *Assume that $\to_a$ and $\to_b$ are confluent. Then, if $\to_a$ commutes with $\to_b$, then $\to_a \cup \to_b$ is confluent.*

Lemma 3.3.8 (Akama [Aka93])**.** *Let $\to_a$ and $\to_b$ be two confluent and terminating relations. Then if $\to_a$ commutes with $\to_b$-normal forms, then $\to_a \cup \to_b$ is also a confluent and terminating relation.*

We introduce a diagram, introduced by Di Cosmo, Piperno and Geser [Cos96]. The diagram and the related properties are prefixed with the prefix DPG.

Definition 3.3.10 (DPG-commutation)**.** We say that a relation $\to_a$ DPG-commutes with $\to_b$ if the following diagram holds:



Or, formally, for every x, y_a and y_b, if $x \to_a y_a$ and $x \to_b y_b$, then there exists z such that $y_a \to_b^\star z$ and $y_b \to_a^+ z$.

Under a condition on termination, the DPG diagram directly permits to prove the commutation of two relations:

Lemma 3.3.9 (DPG-commutation condition)**.** *Assume that $\to_a$ is a terminating relation. If $\to_a$ DPG-commutes with $\to_b$, then $\to_a^\star$ commutes with $\to_b^\star$.*

It has been demonstrated [Cos96] that the above lemma is useful in many cases, since it applies to a large spectrum of relations, where Lemma 3.3.4 cannot be applied.

The DPG diagram also permits to show commutation with normal forms:

Lemma 3.3.10 (Condition for commutation with normal forms)**.** *Assume that $\to_a$ DPG-commutes with $\to_b$ and that $\to_a$ is a terminating relation. If $\to_b$ has unique normal forms and if $\to_a$ preserves $\to_b$-normal forms, then $\to_a$ commutes with $\to_b$ normal forms.*

The two previous lemmas permit to prove the enhanced versions of Hindley-Rosen's and Akama's lemmas.

Lemma 3.3.11 (DPG-Hindley-Rosen)**.** *Assume that $\to_a$ and $\to_b$ are two confluent relations. If $\to_a$ DPG-commutes with $\to_b$, and if $\to_a$ is terminating, then $\to_a \cup \to_b$ is confluent.*

Lemma 3.3.12 (DPG-Akama)**.** *Let $\to_a$ and $\to_b$ be two confluent and terminating relations. If $\to_a$ DPG-commutes with $\to_b$, and if $\to_a$ preserves $\to_b$-normal forms, then $\to_a \cup \to_b$ is also a confluent and terminating relation.*

The use of the DPG commutation with a condition on termination is a special case of the general technique of Van Oostrom's decreasing diagrams [vO08], which we were not aware of at the beginning of this study. We do not know whether they have been mechanically verified: we think that it would provide a very valuable and powerful toolbox. The above lemmas are sufficient for us to prove the confluence and normalization results we want in the rest of the chapter.

3.4 Warm-up: the simply-typed case

This section presents our technique of explicit expansions in the case of the simply typed λ-calculus. It is meant to ease the understanding of the technique, since it has easy proofs, whereas the case of the full system with singleton kinds is more involved. We first recall a definition of the simply typed λ-calculus with βη-equality, and a new construct, called *expansor*, that specifies where η-expansion steps can be performed. We also introduce a reduction semantics for this extended language and show it enjoys confluence and strong normalization. We then show that, up to the insertion of enough expansors, βη-equality is characterized by the reflexive, symmetric, transitive closure of our reduction relation. Lemma 3.4.22 on page 73 constitutes an essential result for completeness.

3.4.1 Definitions

The syntax of the simply typed λ-calculus with pairs is recalled below. Notice that we operate the same shift as before: we talk about types and kinds, instead of terms and types, but this is, again, only a cosmetic detail, which should not bother the reader.

Definition 3.4.1 (Kinds and types)**.**

$$\begin{array}{rcl} \kappa & ::= & \star \mid \kappa \to \kappa \mid \kappa \times \kappa \\ \tau & ::= & \alpha \mid \lambda(\alpha :: \kappa)\,\tau \mid \tau\,\tau \mid (\tau,\tau) \mid \tau.\mathrm{i} \mid \eta_\kappa \end{array}$$

The syntax of kinds and types are as usual for the simply typed λ-calculus, except the fact that we add a family of constants η_κ, that are indexed by kinds. We call the constant η_κ the *expansor* at kind κ. The intuition is that, when applied to a type τ (of kind κ), it will reduce to the η-expansion of τ at the kind κ.

The binding structure of terms and their free type variables are the usual one: notice that $\mathrm{fv}(\eta_\kappa) \triangleq \emptyset$, which is consistent with the presentation of expansors as constants. Substitution is also defined the usual way, and, unsurprisingly, $\eta_\kappa[\alpha \leftarrow \tau] \triangleq \eta_\kappa$ holds.

We recall the typing rules for the simply typed λ-calculus. The typing environments are composed of distinct bindings, and the typing judgment for the simply typed λ-calculus has been extended with a typing rule for expansors: η_κ have kind $\kappa \to \kappa$.

Definition 3.4.2 (Wellformed environments)**.**

$$\textsc{WfEnvEmpty}\ \frac{}{\varepsilon \vdash_{\mathsf{ST}} \mathrm{ok}} \qquad \textsc{WfEnvVar}\ \frac{\Gamma \vdash_{\mathsf{ST}} \mathrm{ok} \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \alpha :: \kappa \vdash_{\mathsf{ST}} \mathrm{ok}}$$

Definition 3.4.3 (Wellformed types)**.**

$$\textsc{Var}\ \frac{\alpha :: \kappa \in \Gamma \qquad \Gamma \vdash_{\mathsf{ST}} \mathrm{ok}}{\Gamma \vdash_{\mathsf{ST}} \alpha :: \kappa} \qquad \textsc{Const}\ \frac{\Gamma \vdash_{\mathsf{ST}} \mathrm{ok}}{\Gamma \vdash_{\mathsf{ST}} \eta_\kappa :: \kappa \to \kappa} \qquad \textsc{Lam}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{ST}} \tau :: \kappa_2}{\Gamma \vdash_{\mathsf{ST}} \lambda(\alpha :: \kappa_1)\,\tau :: \kappa_1 \to \kappa_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash_{\mathsf{ST}} \tau_1 :: \kappa_2 \to \kappa_1 \qquad \Gamma \vdash_{\mathsf{ST}} \tau_2 :: \kappa_2}{\Gamma \vdash_{\mathsf{ST}} \tau_1\,\tau_2 :: \kappa_1} \qquad \textsc{Pair}\ \frac{\Gamma \vdash_{\mathsf{ST}} \tau_1 :: \kappa_1 \qquad \Gamma \vdash_{\mathsf{ST}} \tau_2 :: \kappa_2}{\Gamma \vdash_{\mathsf{ST}} (\tau_1,\tau_2) :: \kappa_1 \times \kappa_2} \qquad \textsc{Proj}\ \frac{\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa_1 \times \kappa_2}{\Gamma \vdash_{\mathsf{ST}} \tau.\mathrm{i} :: \kappa_\mathrm{i}}$$

Evaluation contexts are all contexts with one hole: they do not restrict reduction.

Definition 3.4.4 (Evaluation contexts)**.**

$$\mathcal{C} \quad ::= \quad [\cdot] \mid \lambda(\alpha :: \kappa)\,\mathcal{C} \mid \tau\,\mathcal{C} \mid \mathcal{C}\,\tau \mid (\mathcal{C},\tau) \mid (\tau,\mathcal{C}) \mid \mathcal{C}.\mathrm{i}$$

The reduction relation is defined as follows.

Definition 3.4.5 (Reduction)**.**

$$\begin{array}{rcl} (\lambda(\alpha :: \kappa)\,\tau_1)\,\tau_2 & \xrightarrow{\beta} & \tau_1[\alpha \leftarrow \tau_2] \\ (\tau_1,\tau_2).\mathrm{i} & \xrightarrow{\pi} & \tau_\mathrm{i} \\ \eta_\star & \xrightarrow{\eta_\star} & \lambda(\alpha :: \star)\,\alpha \\ \eta_{\kappa_1 \to \kappa_2} & \xrightarrow{\eta_\to} & \lambda(\mathrm{f} :: \kappa_1 \to \kappa_2)\,\lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\,(\mathrm{f}\,(\eta_{\kappa_1}\,\alpha)) \\ \eta_{\kappa_1 \times \kappa_2} & \xrightarrow{\eta_\times} & \lambda(\mathrm{p} :: \kappa_1 \times \kappa_2)\,((\eta_{\kappa_1}\,(\mathrm{p}.1))\,,\eta_{\kappa_2}\,(\mathrm{p}.2)) \\ \text{if } M & \longrightarrow & M', \text{ then } \mathcal{C}[M] \longrightarrow \mathcal{C}[M'] \end{array}$$

$$\textsc{EqBeta}\ \frac{\Gamma \vdash_{\mathsf{ST}} (\lambda(\alpha :: \kappa)\, \tau_1)\ \tau_2 :: \kappa'}{\Gamma \vdash_{\beta\pi\eta} (\lambda(\alpha :: \kappa)\, \tau_1)\ \tau_2 \equiv \tau_1[\alpha \leftarrow \tau_2]} \qquad \textsc{EqPi}\ \frac{\Gamma \vdash_{\mathsf{ST}} (\tau_1, \tau_2).\mathrm{i} :: \kappa' \qquad \mathrm{i} \in \{1,2\}}{\Gamma \vdash_{\beta\pi\eta} (\tau_1, \tau_2).\mathrm{i} \equiv \tau_{\mathrm{i}}}$$

$$\textsc{EqEtaArrow}\ \frac{\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa \to \kappa' \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma \vdash_{\beta\pi\eta} \tau \equiv \lambda(\alpha :: \kappa)\, \tau\ \alpha} \qquad \textsc{EqEtaProd}\ \frac{\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa_1 \times \kappa_2}{\Gamma \vdash_{\beta\pi\eta} \tau \equiv (\tau.1, \tau.2)} \qquad \textsc{EqVar}\ \frac{\alpha :: \kappa \in \Gamma \qquad \Gamma \vdash_{\mathsf{ST}} \mathrm{ok}}{\Gamma \vdash_{\beta\pi\eta} \alpha \equiv \alpha}$$

$$\textsc{EqExpansor}\ \frac{\Gamma \vdash_{\mathsf{ST}} \mathrm{ok}}{\Gamma \vdash_{\beta\pi\eta} \eta_\kappa \equiv \eta_\kappa} \qquad \textsc{EqLam}\ \frac{\Gamma, \alpha :: \kappa \vdash_{\beta\pi\eta} \tau_1 \equiv \tau_2}{\Gamma \vdash_{\beta\pi\eta} \lambda(\alpha :: \kappa)\, \tau_1 \equiv \lambda(\alpha :: \kappa)\, \tau_2} \qquad \textsc{EqApp}\ \frac{\Gamma \vdash_{\beta\pi\eta} \tau_1 \equiv \tau_1' \qquad \Gamma \vdash_{\beta\pi\eta} \tau_2 \equiv \tau_2'}{\Gamma \vdash_{\beta\pi\eta} \tau_1\ \tau_2 \equiv \tau_1\ \tau_2}$$

$$\textsc{EqPair}\ \frac{\Gamma \vdash_{\beta\pi\eta} \tau_1 \equiv \tau_1' \qquad \Gamma \vdash_{\beta\pi\eta} \tau_2 \equiv \tau_2'}{\Gamma \vdash_{\beta\pi\eta} (\tau_1, \tau_2) \equiv (\tau_1', \tau_2')} \qquad \textsc{EqProj}\ \frac{\Gamma \vdash_{\beta\pi\eta} \tau \equiv \tau'}{\Gamma \vdash_{\beta\pi\eta} \tau.\mathrm{i} \equiv \tau'.\mathrm{i}} \qquad \textsc{EqSym}\ \frac{\Gamma \vdash_{\beta\pi\eta} \tau' \equiv \tau}{\Gamma \vdash_{\beta\pi\eta} \tau \equiv \tau'}$$

$$\textsc{EqTrans}\ \frac{\Gamma \vdash_{\beta\pi\eta} \tau \equiv \tau' \qquad \Gamma \vdash_{\beta\pi\eta} \tau' \equiv \tau''}{\Gamma \vdash_{\beta\pi\eta} \tau \equiv \tau''}$$

Figure 3.11: $\beta\pi\eta$-equality.

We define a small-step reduction relation as the closure of head reduction under evaluation contexts. Head reduction is composed of the contraction of redexes for applications and pairs, to which we add the reduction of expansors: the reduction of η_κ is directed by the kind κ, and produces a function that η-expands its argument at the kind κ. Expanding at the base kind does nothing, so $\eta_\star$ reduces to the identity. Expanding τ at an arrow kind $\eta_{\kappa_1 \to \kappa_2}$ produces a function that returns a function whose body is the expansion of the application of the first argument applied to the expansion of the second argument. Similarly, $\eta_{\kappa_1 \times \kappa_2}$ reduces to a function that returns the pair of the expansions of its projections. Head reduction is then closed under evaluation contexts. Note that we might drop the label on the reduction arrow in the rest of the chapter. Conversely, we might add a label on the arrow when reducing in an arbitrary context, to indicate which head reduction rule has been used.

Definition 3.4.6 ($\eta_\bullet$-reduction)**.** We define $\eta_\bullet$-reduction as the subset of reduction that deals with expansors, that is, as the relation $\xrightarrow{\eta_\star} \cup \xrightarrow{\eta_\times} \cup \xrightarrow{\eta_\to}$ that is closed under contexts.

In Figure 3.11, we define $\beta\pi\eta$-equality as the closure under congruence, symmetry and transitivity of the β and η rules on functions and pairs, restricted to wellformed types. In this definition, we consider the expansors as constants. For instance, it is false that $\vdash_{\beta\pi\eta} \eta_\star \equiv \lambda(\alpha :: \star)\, \alpha$.

We can erase expansors by replacing them with the identity, as defined below. Erasure permits to trivially inject the language into the simply typed λ-calculus.

Definition 3.4.7 (Erasure of expansors)**.**

$$\begin{array}{rcl}
\lfloor \alpha \rfloor & \triangleq & \alpha \\
\lfloor \eta_\kappa \rfloor & \triangleq & \lambda(\alpha :: \kappa)\, \alpha \\
\lfloor (\lambda(\alpha :: \kappa)\, \tau) \rfloor & \triangleq & \lambda(\alpha :: \kappa)\, \lfloor \tau \rfloor \\
\lfloor (\tau_1\ \tau_2) \rfloor & \triangleq & \lfloor \tau_1 \rfloor\ \lfloor \tau_2 \rfloor \\
\lfloor (\tau_1, \tau_2) \rfloor & \triangleq & (\lfloor \tau_1 \rfloor, \lfloor \tau_2 \rfloor) \\
\lfloor (\tau.\mathrm{i}) \rfloor & \triangleq & \lfloor \tau \rfloor.\mathrm{i}
\end{array}$$

We define a way to insert expansors: $\lceil \tau \rceil^\Gamma$ replaces every variable α (free or bound) by their expanded versions $\eta_\kappa\ \alpha$, where κ is the kind that is assigned to α by the current environment Γ.

Definition 3.4.8 (Insertion of expansors)**.**

$$\begin{array}{rcl}
\lceil \alpha \rceil^\Gamma & \triangleq & \begin{cases} \eta_{\Gamma(\alpha)}\ \alpha & \text{if } \alpha \in \operatorname{dom}\Gamma \\ \alpha & \text{otherwise} \end{cases} \\
\lceil \eta_\kappa \rceil^\Gamma & \triangleq & \eta_\kappa \\
\lceil \lambda(\alpha :: \kappa)\, \tau \rceil^\Gamma & \triangleq & \lambda(\alpha :: \kappa)\, \lceil \tau \rceil^{\Gamma, \alpha :: \kappa} \\
\lceil \tau_1\ \tau_2 \rceil^\Gamma & \triangleq & \lceil \tau_1 \rceil^\Gamma\ \lceil \tau_2 \rceil^\Gamma \\
\lceil (\tau_1, \tau_2) \rceil^\Gamma & \triangleq & (\lceil \tau_1 \rceil^\Gamma, \lceil \tau_2 \rceil^\Gamma) \\
\lceil \tau.\mathrm{i} \rceil^\Gamma & \triangleq & \lceil \tau \rceil^\Gamma.\mathrm{i}
\end{array}$$

3.4.2 Subject reduction

Theorem 3.4.1 (Subject reduction)**.** *If* $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$ *and* $\tau \xrightarrow{\beta\pi\eta_\bullet} \tau'$*, then* $\Gamma \vdash_{\mathsf{ST}} \tau' :: \kappa$*.*

Proof. Subject reduction for $\beta\pi$ is a well-known result. The case of $\eta_\bullet$ is immediate. □

The erasure and insertion of expansors preserve wellformedness.

Lemma 3.4.2. $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$ *holds iff* $\Gamma \vdash_{\mathsf{ST}} \lfloor \tau \rfloor :: \kappa$ *holds.*

Lemma 3.4.3. $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$ *holds iff* $\Gamma \vdash_{\mathsf{ST}} \lceil \tau \rceil^\Gamma :: \kappa$ *holds.*

3.4.3 Confluence and strong normalization

In this section, we establish the confluence and strong normalization properties of our reduction relation in a modular way, using the results that have been introduced in Section 3.3 on page 65.

It is already a well-known result that $\beta\pi$-reduction is confluent.

Theorem 3.4.4. $\xrightarrow{\beta\pi}$ *is confluent.*

We can also prove by induction the diamond property for $\eta_\bullet$-reduction.

Lemma 3.4.5. $\xrightarrow{\eta_\bullet}$ *has the diamond property.*

It follows from Lemma 3.3.3 on page 66 that $\eta_\bullet$-reduction is a confluent relation.

Proposition 3.4.6. $\xrightarrow{\eta_\bullet}$ *is confluent.*

We recall the strong normalization property for the simply typed λ-calculus. See for instance [GTL89] for a detailed proof.

Theorem 3.4.7. $\xrightarrow{\beta\pi}$ *is strongly normalizing on wellformed types.*

We can easily establish that $\eta_\bullet$-reduction is also strongly normalizing, by considering the multiset of the kinds of the expansors as a measure: $\eta_\bullet$-reduction rules either discard the expansors, or replace them with expansors at a strictly (structurally) smaller kind.

Proposition 3.4.8. *$\xrightarrow{\eta_\bullet}$ is strongly normalizing.*

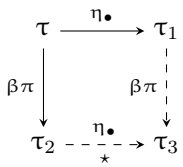
Since we have established the normalization and confluence properties separately for $\beta\pi$ and for $\eta_\bullet$, we now get down to their combination into the full reduction relation. Firstly, $\eta_\bullet$-reduction commutes with substitution as follows:

Lemma 3.4.9. *The following assertions hold:*

- *If* $\tau_1 \xrightarrow{\eta_\bullet} \tau_1'$ *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta_\bullet} \tau_1'[\alpha \leftarrow \tau_2]$.
- *If* $\tau_2 \xrightarrow{\eta_\bullet} \tau_2'$ *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta_\bullet}{}^{\star} \tau_1[\alpha \leftarrow \tau_2']$.

This result on substitution and $\eta_\bullet$-reduction permits to prove a commutation property for the two sub-relations.

Lemma 3.4.10. *The following commutation diagram holds:*



Lemma 3.4.11. *$\xrightarrow{\beta\pi}$ DPG-commutes with $\xrightarrow{\eta_\bullet}$.*

Proof. Consequence of Lemma 3.4.10. □

Finally, we can prove that the whole relation is confluent on wellformed types.

Theorem 3.4.12 (Confluence)**.** *The relation $\xrightarrow{\beta\pi\eta_\bullet}$ is confluent on wellformed types.*

Proof. By Lemma 3.3.11 on page 67, using Theorem 3.4.4, Proposition 3.4.6 and Theorem 3.4.1. □

We now focus on the normalization property for the reduction relation.

Lemma 3.4.13. *The relation $\xrightarrow{\beta\pi}$ preserves $\xrightarrow{\eta_\bullet}$-normal forms.*

As a consequence, the whole reduction is strongly normalizing on wellformed types.

Theorem 3.4.14 (Strong normalization)**.** *The reduction relation $\xrightarrow{\beta\pi\eta_\bullet}$ is strongly normalizing on wellformed types.*

Proof. Consequence of Lemma 3.3.12 on page 67, using Theorem 3.4.7, and Proposition 3.4.8, and Lemma 3.4.13, and Theorem 3.4.1. □

We have established that our reduction relation is confluent and strongly normalizing on wellformed types. The diagrams from Section 3.3 on page 65 greatly simplified the proof. It is certainly possible that it is confluent for arbitrary types, for instance using the technique of parallel reductions [Tak95], but it is not clear how to prove confluence in a modular way.

3.4.4 Adequacy

In this section, we show that $\beta\pi\eta$-equality reduces to $\beta\pi\eta_\bullet$-equality up to the insertion of expansors. It provides a simple way to test for $\beta\pi\eta$-equality. We begin with properties of expansors: first, expansors are idempotent.

Lemma 3.4.15 (Idempotency of expansors)**.** *For all kind* κ *and type* τ, $\eta_\kappa\ (\eta_\kappa\ \tau) \overset{\beta\pi\eta_\bullet}{\longleftrightarrow} \eta_\kappa\ \tau$ *holds.*

Proof. By induction on κ:

- $\kappa = \star$:
$$\eta_\star\ (\eta_\star\ \tau) \xrightarrow{\eta_\star} (\lambda(\alpha :: \star)\,\alpha)\ (\eta_\star\ \tau) \xrightarrow{\beta} \eta_\star\ \tau$$

- $\kappa = \kappa_1 \to \kappa_2$:
$$\begin{array}{ll}
 & \eta_{\kappa_1\to\kappa_2}\ (\eta_{\kappa_1\to\kappa_2}\ \tau) \\
\xrightarrow{\eta_\to} & (\lambda(f :: \kappa_1 \to \kappa_2)\,\lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ (f\ (\eta_{\kappa_1}\ \alpha)))\ (\eta_{\kappa_1\to\kappa_2}\ \tau) \\
\xrightarrow{\beta} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ ((\eta_{\kappa_1\to\kappa_2}\ \tau)\ (\eta_{\kappa_1}\ \alpha)) \\
\xrightarrow{\eta_\to} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ (((\lambda(f :: \kappa_1 \to \kappa_2)\,\lambda(\beta :: \kappa_1)\,\eta_{\kappa_2}\ (f\ (\eta_{\kappa_1}\ \beta)))\ \tau)\ (\eta_{\kappa_1}\ \alpha)) \\
\xrightarrow{\beta} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ ((\lambda(\beta :: \kappa_1)\,\eta_{\kappa_2}\ (\tau\ (\eta_{\kappa_1}\ \beta)))\ (\eta_{\kappa_1}\ \alpha)) \\
\xrightarrow{\beta} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ (\eta_{\kappa_2}\ (\tau\ (\eta_{\kappa_1}\ (\eta_{\kappa_1}\ \alpha)))) \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ (\eta_{\kappa_2}\ (\tau\ (\eta_{\kappa_1}\ \alpha))) & \text{by induction hypothesis} \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2}\ (\tau\ (\eta_{\kappa_1}\ \alpha)) & \text{by induction hypothesis} \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & \eta_{\kappa_1\to\kappa_2}\ \tau
\end{array}$$

- $\kappa = \kappa_1 \times \kappa_2$:
$$\begin{array}{ll}
 & \eta_{\kappa_1\times\kappa_2}\ (\eta_{\kappa_1\times\kappa_2}\ \tau) \\
\xrightarrow{\eta_\times} & \eta_{\kappa_1\times\kappa_2}\ ((\lambda(p :: \kappa_1 \times \kappa_2)\,(\eta_{\kappa_1}\ (p.1), \eta_{\kappa_2}\ (p.2)))\ \tau) \\
\xrightarrow{\beta} & \eta_{\kappa_1\times\kappa_2}\ (\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)) \\
\xrightarrow{\eta_\times} & (\lambda(p :: \kappa_1 \times \kappa_2)\,(\eta_{\kappa_1}\ (p.1), \eta_{\kappa_2}\ (p.2)))\ (\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)) \\
\xrightarrow{\beta} & (\eta_{\kappa_1}\ ((\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)).1), \eta_{\kappa_2}\ ((\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)).2)) \\
\xrightarrow{\pi} & (\eta_{\kappa_1}\ (\eta_{\kappa_1}\ (\tau.1)), \eta_{\kappa_2}\ ((\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)).2)) \\
\xrightarrow{\pi} & (\eta_{\kappa_1}\ (\eta_{\kappa_1}\ (\tau.1)), \eta_{\kappa_2}\ (\eta_{\kappa_2}\ (\tau.2))) \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & (\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\eta_{\kappa_2}\ (\tau.2))) & \text{by induction hypothesis} \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & (\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2}\ (\tau.2)) & \text{by induction hypothesis} \\
\overset{\beta\pi\eta_\bullet}{\longleftrightarrow} & \eta_{\kappa_1\times\kappa_2}\ \tau
\end{array}$$

□

We then show that expanding an expansor adds no information.

Lemma 3.4.16 (Expansion of expansors)**.** *For every kind* κ, $\eta_\kappa \overset{\beta\pi\eta_\bullet}{\longleftrightarrow} \eta_{\kappa\to\kappa}\ \eta_\kappa$.

Proof. We first notice that $\eta_{\kappa\to\kappa}\ \eta_\kappa \overset{\beta\pi\eta_\bullet}{\longleftrightarrow} \lambda(\alpha :: \kappa)\,\eta_\kappa\ \alpha$ thanks to Lemma 3.4.15. It suffices to show that $\lambda(\alpha :: \kappa)\,\eta_\kappa\ \alpha \overset{\beta\pi\eta_\bullet}{\longleftrightarrow} \eta_\kappa$. We proceed by case analysis on κ. Each case is easily solved by a few rewrite steps.

- $\underline{\kappa = \star}$:

$$\lambda(\alpha :: \star)\, \eta_\star\ \alpha \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lambda(\alpha :: \star)\, \alpha \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_\star$$

- $\underline{\kappa = \kappa_1 \to \kappa_2}$:

$$\lambda(\alpha :: \kappa_1 \to \kappa_2)\, \eta_{\kappa_1 \to \kappa_2}\ \alpha \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lambda(\alpha :: \kappa_1 \to \kappa_2)\, \lambda(\beta :: \kappa_1)\, \eta_{\kappa_2}\ (\alpha\ (\eta_{\kappa_1}\ \beta)) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_{\kappa_1 \to \kappa_2}$$

- $\underline{\kappa = \kappa_1 \times \kappa_2}$:

$$\lambda(\alpha :: \kappa_1 \times \kappa_2)\, \eta_{\kappa_1 \times \kappa_2}\ \alpha \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lambda(\alpha :: \kappa_1 \times \kappa_2)\, (\eta_{\kappa_1}\ \alpha.1, \eta_{\kappa_2}\ \alpha.2) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_{\kappa_1 \times \kappa_2}$$ □

We go on with properties of erasure and insertion of expansors. First, these operations are stable under renamings.

Lemma 3.4.17. *For all* Γ, Γ', α, β, κ, τ, *if* $\alpha \notin \operatorname{dom}\Gamma'$ *and* $\beta \notin \operatorname{dom}\Gamma, \Gamma'$ *then* $\lceil\tau\rceil^{\Gamma,\alpha::\kappa,\Gamma'}[\alpha \leftarrow \beta] = \lceil\tau[\alpha \leftarrow \beta]\rceil^{\Gamma,\beta::\kappa,\Gamma'}$.

Then, the operations are idempotent, and commute with substitution as follows.

Lemma 3.4.18. *The following assertions hold:*

- $\lfloor\lfloor\tau\rfloor\rfloor = \lfloor\tau\rfloor$;
- $\lceil\lceil\tau\rceil^\Gamma\rceil^\Gamma \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\tau\rceil^\Gamma$;
- $\lfloor\tau_1[\alpha \leftarrow \tau_2]\rfloor = \lfloor\tau_1\rfloor[\alpha \leftarrow \lfloor\tau_2\rfloor]$;
- *If* $\alpha \notin \operatorname{dom}\Gamma'$, *then* $\lceil\tau_1\rceil^{\Gamma,\alpha::\kappa,\Gamma'}[\alpha \leftarrow \lceil\tau_2\rceil^{\Gamma,\Gamma'}] = \lceil\tau_1[\alpha \leftarrow \eta_\kappa\ \tau_2]\rceil^{\Gamma,\Gamma'}$;
- *If* $\alpha \notin \operatorname{dom}\Gamma, \Gamma'$, *then* $\lceil\tau_1\rceil^{\Gamma,\Gamma'}[\alpha \leftarrow \lceil\tau_2\rceil^{\Gamma,\Gamma'}] = \lceil\tau_1[\alpha \leftarrow \tau_2]\rceil^{\Gamma,\Gamma'}$.

The items that deal with idempotency are consequences of Lemma 3.4.15. We notice that insertion of expansors does not always commute with substitution: it only commutes when the variable to be substituted is not in the environment; otherwise, an expansor is inserted at every substitution site.

From Lemma 3.4.15, one can also show that adding expansors at variables that will be expanded has no effect.

Lemma 3.4.19. *Assume* $\alpha \notin \operatorname{dom}\Gamma'$. *Then* $\lceil\tau\rceil^{\Gamma,\alpha::\kappa,\Gamma'} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\tau[\alpha \leftarrow \eta_\kappa\ \alpha]\rceil^{\Gamma,\alpha::\kappa,\Gamma'}$ *holds.*

We now prove that convertibility is sound with respect to $\beta\pi\eta$-equality.

Lemma 3.4.20. *If* $\Gamma \vdash_{\mathsf{ST}} \tau_1 :: \kappa$ *and* $\tau_1 \overset{\beta\pi\eta^\bullet}{\longrightarrow} \tau_2$ *then* $\Gamma \vdash_{\beta\pi\eta} \lfloor\tau_1\rfloor \equiv \lfloor\tau_2\rfloor$.

Proof. The result for the cases β and π is easily proved using Lemma 3.4.18 and contraction rules. In the case of $\eta_\bullet$, it follows directly from extensional rules. □

Proposition 3.4.21 (Soundness)**.** *If* $\Gamma \vdash_{\mathsf{ST}} \tau_1 :: \kappa$, $\Gamma \vdash_{\mathsf{ST}} \tau_2 :: \kappa$ *and* $\tau_1 \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \tau_2$ *then* $\Gamma \vdash_{\beta\pi\eta} \lfloor\tau_1\rfloor \equiv \lfloor\tau_2\rfloor$.

Proof. By induction on convertibility, using Lemma 3.4.20 and Theorem 3.4.1. □

We now focus on a completeness result for our convertibility relation. We first prove that adding an expansor in front of a full insertion of expansors is superfluous on wellformed types. This lemma is the central lemma for the completeness result.

Lemma 3.4.22. *For every* κ, τ, Γ, *if* $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$ *holds, then* $\lceil\tau\rceil^\Gamma \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_\kappa\ \lceil\tau\rceil^\Gamma$ *holds.*

Proof. By induction on the typing judgment:

- VAR case:

$$\begin{array}{rcll} \lceil \alpha \rceil^{\Gamma} & = & \eta_{\kappa}\ \alpha & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \eta_{\kappa}\ (\eta_{\kappa}\ \alpha) & \text{by Lemma 3.4.15} \\ & = & \eta_{\kappa}\ \lceil \alpha \rceil^{\Gamma} & \end{array}$$

- CONST case:

$$\begin{array}{rcll} \lceil \eta_{\kappa_0} \rceil^{\Gamma} & = & \eta_{\kappa_0} & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \eta_{\kappa_0 \to \kappa_0}\ \eta_{\kappa_0} & \text{by Lemma 3.4.16} \\ & = & \eta_{\kappa_0 \to \kappa_0}\ \lceil \eta_{\kappa_0} \rceil^{\Gamma} & \end{array}$$

- LAM case:

$$\begin{array}{rcll} \lceil \lambda(\alpha :: \kappa)\, \tau \rceil^{\Gamma} & = & \lambda(\alpha :: \kappa)\, \lceil \tau \rceil^{\Gamma, \alpha :: \kappa} & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \lambda(\alpha :: \kappa)\, \eta_{\kappa'}\ \lceil \tau \rceil^{\Gamma, \alpha :: \kappa} & \text{by induction hypothesis} \\ & = & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ \lceil \tau \rceil^{\Gamma, \alpha :: \kappa}[\alpha \leftarrow \beta] & \text{by } \alpha\text{-equivalence} \\ & & & \text{for } \beta \notin \{\alpha\} \cup \operatorname{dom}\Gamma \cup \operatorname{fv}(\tau) \\ & = & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ \lceil \tau[\alpha \leftarrow \beta] \rceil^{\Gamma, \beta :: \kappa} & \text{by Lemma 3.4.17} \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ \lceil \tau[\alpha \leftarrow \eta_{\kappa}\ \beta] \rceil^{\Gamma, \beta :: \kappa} & \text{by Lemma 3.4.19} \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ \lceil \tau \rceil^{\Gamma, \beta :: \kappa, \alpha :: \kappa}[\alpha \leftarrow \lceil \beta \rceil^{\Gamma, \beta :: \kappa}] & \text{by Lemma 3.4.18} \\ & = & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ \lceil \tau \rceil^{\Gamma, \beta :: \kappa, \alpha :: \kappa}[\alpha \leftarrow \eta_{\kappa}\ \beta] & \\ & \overset{\beta}{\longleftrightarrow} & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ ((\lambda(\alpha :: \kappa)\, \lceil \tau \rceil^{\Gamma, \beta :: \kappa, \alpha :: \kappa})\ (\eta_{\kappa}\ \beta)) & \\ & = & \lambda(\beta :: \kappa)\, \eta_{\kappa'}\ (\lceil \lambda(\alpha :: \kappa)\, \tau \rceil^{\Gamma, \beta :: \kappa}\ (\eta_{\kappa}\ \beta)) & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \eta_{\kappa \to \kappa'}\ \lceil \lambda(\alpha :: \kappa)\, \tau \rceil^{\Gamma} & \end{array}$$

- APP case:

$$\begin{array}{rcll} \lceil \tau_1\ \tau_2 \rceil^{\Gamma} & = & \lceil \tau_1 \rceil^{\Gamma}\ \lceil \tau_2 \rceil^{\Gamma} & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & (\eta_{\kappa_2 \to \kappa_1}\ \lceil \tau_1 \rceil^{\Gamma})\ (\eta_{\kappa_2}\ \lceil \tau_2 \rceil^{\Gamma}) & \text{by induction hypothesis} \\ & \overset{\eta^{\bullet}}{\longleftrightarrow} & (\lambda(\alpha :: \kappa_2)\, \eta_{\kappa_1}\ (\lceil \tau_1 \rceil^{\Gamma}\ (\eta_{\kappa_2}\ \alpha)))\ \lceil \tau_2 \rceil^{\Gamma} & \text{with } \alpha \notin \operatorname{fv}(\tau_1) \\ & \overset{\beta}{\longleftrightarrow} & \eta_{\kappa_1}\ (\lceil \tau_1 \rceil^{\Gamma}\ (\eta_{\kappa_2}\ \lceil \tau_2 \rceil^{\Gamma})) & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & \eta_{\kappa_1}\ (\lceil \tau_1 \rceil^{\Gamma}\ \lceil \tau_2 \rceil^{\Gamma}) & \text{by induction hypothesis} \\ & = & \eta_{\kappa_1}\ \lceil \tau_1\ \tau_2 \rceil^{\Gamma} & \end{array}$$

- PAIR case:

$$\begin{array}{rcll} \lceil (\tau_1, \tau_2) \rceil^{\Gamma} & = & (\lceil \tau_1 \rceil^{\Gamma}, \lceil \tau_2 \rceil^{\Gamma}) & \\ & \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & (\eta_{\kappa_1}\ \lceil \tau_1 \rceil^{\Gamma}, \eta_{\kappa_2}\ \lceil \tau_2 \rceil^{\Gamma}) & \text{by induction hypothesis} \\ & \overset{\pi}{\longleftrightarrow} & (\eta_{\kappa_1}\ ((\lceil \tau_1 \rceil^{\Gamma}, \lceil \tau_2 \rceil^{\Gamma}).1), \eta_{\kappa_2}\ ((\lceil \tau_1 \rceil^{\Gamma}, \lceil \tau_2 \rceil^{\Gamma}).2)) & \\ & \overset{\beta}{\longleftrightarrow} & (\lambda(p :: \kappa_1 \times \kappa_2)\, (\eta_{\kappa_1}\ (p.1), \eta_{\kappa_2}\ (p.2)))\ (\lceil \tau_1 \rceil^{\Gamma}, \lceil \tau_2 \rceil^{\Gamma}) & \\ & \overset{\eta^{\bullet}}{\longleftrightarrow} & \eta_{\kappa_1 \times \kappa_2}\ (\lceil \tau_1 \rceil^{\Gamma}, \lceil \tau_2 \rceil^{\Gamma}) & \\ & = & \eta_{\kappa_1 \times \kappa_2}\ \lceil (\tau_1, \tau_2) \rceil^{\Gamma} & \end{array}$$

- Proj case:

$$\begin{array}{rcll}
\lceil \tau.\mathfrak{i} \rceil^{\Gamma} & = & \lceil \tau \rceil^{\Gamma}.\mathfrak{i} & \\
& \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} & (\eta_{\kappa_1 \times \kappa_2}\ \lceil \tau \rceil^{\Gamma}).\mathfrak{i} & \text{by induction hypothesis} \\
& \overset{\eta^{\bullet}}{\longleftrightarrow} & ((\lambda(p :: \kappa_1 \times \kappa_2)\,(\eta_{\kappa_1}\ (p.1), \eta_{\kappa_2}\ (p.2)))\ \lceil \tau \rceil^{\Gamma}).\mathfrak{i} & \\
& \overset{\beta}{\longleftrightarrow} & (\eta_{\kappa_1}\ (\lceil \tau \rceil^{\Gamma}.1), \eta_{\kappa_2}\ (\lceil \tau \rceil^{\Gamma}.2)).\mathfrak{i} & \\
& \overset{\pi}{\longleftrightarrow} & \eta_{\kappa_i}\ (\lceil \tau \rceil^{\Gamma}.\mathfrak{i}) & \\
& = & \eta_{\kappa_i}\ \lceil \tau.\mathfrak{i} \rceil^{\Gamma} &
\end{array}$$

□

It follows from the previous lemma that a wellformed type where expansors are fully inserted is convertible with its η-expansion.

Lemma 3.4.23. *For every* κ, τ, Γ, *such that* $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$, *the following assertions hold:*

- *if* $\kappa = \kappa_1 \to \kappa_2$: $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} \lambda(\alpha :: \kappa_1)\ \lceil \tau \rceil^{\Gamma}\ \alpha$, *provided* $\alpha \notin \mathrm{fv}(\tau)$;
- *if* $\kappa = \kappa_1 \times \kappa_2$: $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} (\lceil \tau \rceil^{\Gamma}.1, \lceil \tau \rceil^{\Gamma}.2)$.

Proof. By induction on κ, using Lemma 3.4.22. □

Proposition 3.4.24 (Completeness)**.** *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *holds, then* $\lceil \tau_1 \rceil^{\Gamma} \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} \lceil \tau_2 \rceil^{\Gamma}$ *holds.*

Proof. By induction on the equality judgment. The η-rules are solved using Lemma 3.4.23; the π-rule is straightforward; the β-rule is solved by Lemma 3.4.22 and Lemma 3.4.18; other rules are solved by induction. □

Theorem 3.4.25 (Adequacy)**.** *For every* Γ, κ, τ *and* τ', *such that* τ *and* τ' *are expansor-free,* $\Gamma \vdash_{\beta\pi\eta} \tau \equiv \tau'$ *holds iff* $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} \lceil \tau' \rceil^{\Gamma}$ *and* $\Gamma \vdash_{\mathsf{ST}} \tau :: \kappa$ *and* $\Gamma \vdash_{\mathsf{ST}} \tau' :: \kappa$ *hold.*

Proof. By Proposition 3.4.21 and Proposition 3.4.24, using the fact that $\lfloor \tau \rfloor = \tau$ when τ is expansor-free. □

As already mentioned, the essential part of the proof is Lemma 3.4.22. Although it is not visible in the simply typed λ-calculus, this lemma works thanks to the fact that kinds are unique. We will see in the next section, that this lemma generalizes if one considers the minimal kind of the type under consideration.

We announced in Section 3.2 on page 64 that our characterization of equivalence differs from Crary's because we expand every type variable, while he only expands free type variables. We guess that, when prefixed by a head expansion, the two are equivalent.

Conjecture 3.4.1. *If* $\Gamma \vdash \tau :: \kappa$, *then* $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^{\bullet}}{\longleftrightarrow} \eta_{\kappa}\ \tau\{\Gamma\}$, *where* $\{\Gamma\}$ *is the composition of the substitutions* $[\alpha \leftarrow \eta_{\Gamma(\alpha)}\ \alpha]$ *for every* $\alpha \in \mathrm{dom}\,\Gamma$.

However, a proof by induction does not go through: we get stuck in the case of applications, because an extra expansor appears on the argument of the application. We think that the proof could be done with the use of a logical relation. In the case of the system with singleton kinds, indeed, we had the same problem, and it vanished when considering a logical relation (Definition 3.5.15 on page 85) based on the one that Stone and Harper use to prove completeness of their algorithm [SH06].

3.5 Small-step extensional equivalence for singleton types

In this section, we extend the technique from Section 3.4 to the Stone and Harper singleton kind system: we extend their system with explicit expansors, then define a reduction relation, prove its soundness with respect to the kind system and its confluence and strong normalization, and, finally, show the adequacy between convertibility up to this reduction relation and their specification for type equivalence. Normalization will use the same technique of combination of commutation diagrams. The strong normalization property for the β-reduction part of the semantics is based on a translation into the simply typed λ-calculus, that preserves reduction steps and wellformedness. The completeness result of convertibility with respect to type equivalence is the hardest part of this document: it reuses the logical relation used in [SH06].

3.5.1 Definition

We extend the grammar of types with the expansors η_κ that are, again, indexed by their kind.

Definition 3.5.1 (Kinds and types)**.**

$$\begin{array}{rcl} \kappa & ::= & \star \mid \Pi(\alpha:\kappa)\,\kappa \mid \Sigma(\alpha:\kappa)\,\kappa \mid \mathcal{S}(\tau) \\ \tau & ::= & \alpha \mid \lambda(\alpha::\kappa)\,\tau \mid \tau\,\tau \mid (\tau,\tau) \mid \tau.\mathfrak{i} \mid \eta_\kappa \end{array}$$

We define the sub-grammar of normal forms for types and kinds. Unsurprisingly, it relies on the definition of paths.

Definition 3.5.2 (Normal forms)**.**

$$\begin{array}{rcll} \tau^n & ::= & p \mid \lambda(\alpha::\kappa^n)\,\tau^n \mid (\tau^n,\tau^n) & \text{(Normal types)} \\ p & ::= & \alpha \mid p\,\tau^n \mid p.\mathfrak{i} & \text{(Paths)} \\ \kappa^n & ::= & \star \mid \mathcal{S}(\tau^n) \mid \Pi(\alpha:\kappa^n)\,\kappa^n \mid \Sigma(\alpha:\kappa^n)\,\kappa^n & \text{(Normal kinds)} \end{array}$$

We recall the definition of free variables and of capture-avoiding substitution, because of the mutual recursion between types and kinds. It is important to notice that the two functions recurse on the indexes of expansors.

Definition 3.5.3 (Free variables)**.**

$$\begin{array}{rcl} \mathrm{fv}(\alpha) & \triangleq & \{\alpha\} \\ \mathrm{fv}(\eta_\kappa) & \triangleq & \mathrm{fv}(\kappa) \\ \mathrm{fv}(\lambda(\alpha::\kappa)\,\tau) & \triangleq & \mathrm{fv}(\kappa) \cup (\mathrm{fv}(\tau) \setminus \{\alpha\}) \\ \mathrm{fv}(\tau_1\,\tau_2) & \triangleq & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\ \mathrm{fv}((\tau_1,\tau_2)) & \triangleq & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\ \mathrm{fv}(\tau.\mathfrak{i}) & \triangleq & \mathrm{fv}(\tau) \end{array}$$

$$\begin{array}{rcl} \mathrm{fv}(\star) & \triangleq & \{\} \\ \mathrm{fv}(\mathcal{S}(\tau)) & \triangleq & \mathrm{fv}(\tau) \\ \mathrm{fv}(\Pi(\alpha:\kappa_1)\,\kappa_2) & \triangleq & \mathrm{fv}(\kappa_1) \cup (\mathrm{fv}(\kappa_2) \setminus \{\alpha\}) \\ \mathrm{fv}(\Sigma(\alpha:\kappa_1)\,\kappa_2) & \triangleq & \mathrm{fv}(\kappa_1) \cup (\mathrm{fv}(\kappa_2) \setminus \{\alpha\}) \end{array}$$

Definition 3.5.4 (Substitution)**.**

$$
\begin{array}{rcll}
\beta[\alpha \leftarrow \tau] & \triangleq & \begin{cases} \tau & \text{if } \alpha = \beta \\ \beta & \text{otherwise} \end{cases} & \\
\eta_\kappa[\alpha \leftarrow \tau] & \triangleq & \eta_{\kappa[\alpha \leftarrow \tau]} & \\
(\lambda(\beta :: \kappa)\, \tau_1)[\alpha \leftarrow \tau] & \triangleq & \lambda(\beta :: \kappa[\alpha \leftarrow \tau])\, \tau_1[\alpha \leftarrow \tau] & \text{if } \alpha \neq \beta \text{ and } \beta \notin \mathrm{fv}(\tau) \\
(\tau_1\ \tau_2)[\alpha \leftarrow \tau] & \triangleq & \tau_1[\alpha \leftarrow \tau]\ \tau_2[\alpha \leftarrow \tau] & \\
(\tau_1, \tau_2)[\alpha \leftarrow \tau] & \triangleq & (\tau_1[\alpha \leftarrow \tau], \tau_2[\alpha \leftarrow \tau]) & \\
(\tau_1.\mathtt{i})[\alpha \leftarrow \tau] & \triangleq & \tau_1[\alpha \leftarrow \tau].\mathtt{i} & \\
 & & & \\
\star[\alpha \leftarrow \tau] & \triangleq & \star & \\
\mathcal{S}(\tau_1)[\alpha \leftarrow \tau] & \triangleq & \mathcal{S}(\tau_1[\alpha \leftarrow \tau]) & \\
(\Pi(\beta : \kappa_1)\, \kappa_2)[\alpha \leftarrow \tau] & \triangleq & \Pi(\beta : \kappa_1[\alpha \leftarrow \tau])\, \kappa_2[\alpha \leftarrow \tau] & \text{if } \alpha \neq \beta \text{ and } \beta \notin \mathrm{fv}(\tau) \\
(\Sigma(\beta : \kappa_1)\, \kappa_2)[\alpha \leftarrow \tau] & \triangleq & \Sigma(\beta : \kappa_1[\alpha \leftarrow \tau])\, \kappa_2[\alpha \leftarrow \tau] & \text{if } \alpha \neq \beta \text{ and } \beta \notin \mathrm{fv}(\tau)
\end{array}
$$

As in [SH06], our proofs use the following definition of size of kinds. Notice that it does not recurse under singletons. Therefore, the size of a kind is invariant under substitution for any type.

Definition 3.5.5 (Size of a kind)**.**

$$
\begin{array}{rcl}
\mathrm{size}(\star) & \triangleq & 1 \\
\mathrm{size}(\mathcal{S}(\tau)) & \triangleq & 2 \\
\mathrm{size}(\Pi(\alpha : \kappa_1)\, \kappa_2) & \triangleq & 1 + \mathrm{size}(\kappa_1) + \mathrm{size}(\kappa_2) \\
\mathrm{size}(\Sigma(\alpha : \kappa_1)\, \kappa_2) & \triangleq & 1 + \mathrm{size}(\kappa_1) + \mathrm{size}(\kappa_2)
\end{array}
$$

The fact that $\mathrm{size}\,\mathcal{S}(\tau) > \mathrm{size}\,\star$ is used in the definition of the logical relation (Definition 3.5.15 on page 85, Section 3.5.7) on which our completeness lemma is based.

Lemma 3.5.1. *For every* κ, α *and* τ, $\mathrm{size}\,\kappa[\alpha \leftarrow \tau] = \mathrm{size}\,\kappa$.

The reduction contexts are not restricted: in particular, reduction is allowed inside kinds, because they can contain types in singletons; reduction also happens inside the indexes of expansors.

Definition 3.5.6 (Contexts)**.**

$$
\begin{array}{rcl}
\mathcal{C} & ::= & [\cdot] \mid \eta_{\mathcal{K}} \mid \lambda(\alpha :: \mathcal{K})\, \tau \mid \lambda(\alpha :: \kappa)\, \mathcal{C} \\
 & \mid & \tau\ \mathcal{C} \mid \mathcal{C}\ \tau \mid (\mathcal{C}, \tau) \mid (\tau, \mathcal{C}) \mid \mathcal{C}.\mathtt{i} \\
\mathcal{K} & ::= & \Pi(\alpha : \mathcal{K})\, \kappa \mid \Pi(\alpha : \kappa)\, \mathcal{K} \mid \Sigma(\alpha : \mathcal{K})\, \kappa \mid \Sigma(\alpha : \kappa)\, \mathcal{K} \mid \mathcal{S}(\mathcal{C})
\end{array}
$$

As in Section 3.4, the reduction relation is composed of the usual reduction of application and projection redexes on the one hand, and of the reductions of expansors on the other hand.

Definition 3.5.7 (Reduction).

$$
\begin{array}{rcll}
(\lambda(\alpha :: \kappa)\,\tau_1)\;\tau_2 & \xrightarrow{\beta} & \tau_1[\alpha \leftarrow \tau_2] & \\
(\tau_1, \tau_2).\mathrm{i} & \xrightarrow{\pi} & \tau_\mathrm{i} & \\
\eta_\star & \xrightarrow{\eta_\star} & \lambda(\alpha :: \star)\,\alpha & \\
\eta_{\mathcal{S}(\tau_1)} & \xrightarrow{\eta_\mathcal{S}} & \lambda(\alpha :: \star)\,\tau_1 & \text{if } \alpha \notin \mathrm{fv}(\tau_1) \\
\eta_{\Pi(\alpha:\kappa_1)\,\kappa_2} & \xrightarrow{\eta_\Pi} & \lambda(\mathrm{f} :: \Pi(\alpha : \kappa_1)\,\kappa_2)\,\lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\;\alpha]}\;(\mathrm{f}\;(\eta_{\kappa_1}\;\alpha)) & \\
 & & & \text{if } \mathrm{f}, \alpha \notin \mathrm{fv}(\Pi(\alpha : \kappa_1)\,\kappa_2),\ \mathrm{f} \neq \alpha \\
\eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2} & \xrightarrow{\eta_\Sigma} & \lambda(\mathrm{p} :: \Sigma(\alpha : \kappa_1)\,\kappa_2)\,((\eta_{\kappa_1}\;\mathrm{p}.1)\;, \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\;\mathrm{p}.1]}\;\mathrm{p}.2) & \\
 & & & \text{if } \mathrm{p}, \alpha \notin \mathrm{fv}(\Sigma(\alpha : \kappa_1)\,\kappa_2),\ \mathrm{p} \neq \alpha
\end{array}
$$

$$
\frac{\tau \longrightarrow \tau'}{\mathcal{C}[\tau] \longrightarrow \mathcal{C}[\tau']} \qquad \frac{\tau \longrightarrow \tau'}{\mathcal{K}[\tau] \longrightarrow \mathcal{K}[\tau']}
$$

As in Section 3.4.1 on page 68, we may omit the label on the reduction arrow. Remark that, since arrow and pair kinds are dependent, more expansors are inserted than in the case of the simply typed λ-calculus: in particular, expansors are inserted in the kinds that index expansors. As before, we define the sub-relation $\eta_\bullet$ that only deals with the reduction of expansors.

Definition 3.5.8 ($\eta_\bullet$-reduction). We define $\eta_\bullet$-reduction as the subset of reduction that deals with expansors, or, in other words, as the relation $\xrightarrow{\eta_\bullet} \triangleq \xrightarrow{\eta_\star} \cup \xrightarrow{\eta_\Sigma} \cup \xrightarrow{\eta_\mathcal{S}} \cup \xrightarrow{\eta_\Pi}$ that is closed under contexts.

Insertion of expansors generalizes the one of Section 3.4. The main differences are that it recurses on the indexes of expansors, and that we also have to insert expansors in the kinds of the environments.

Definition 3.5.9 (Insertion of expansors).

$$
\begin{array}{rcl}
\lceil \varepsilon \rceil & \triangleq & \varepsilon \\
\lceil \Gamma, \alpha :: \kappa \rceil & \triangleq & \lceil \Gamma \rceil, \alpha :: \lceil \kappa \rceil^{\lceil \Gamma \rceil}
\end{array}
$$

$$
\begin{array}{rcl}
\lceil \star \rceil^\Gamma & \triangleq & \star \\
\lceil \mathcal{S}(\tau) \rceil^\Gamma & \triangleq & \mathcal{S}(\lceil \tau \rceil^\Gamma) \\
\lceil \Pi(\alpha : \kappa_1)\,\kappa_2 \rceil^\Gamma & \triangleq & \Pi(\alpha : \lceil \kappa_1 \rceil^\Gamma)\,\lceil \kappa_2 \rceil^{\Gamma, \alpha :: \lceil \kappa_1 \rceil^\Gamma} \\
\lceil \Sigma(\alpha : \kappa_1)\,\kappa_2 \rceil^\Gamma & \triangleq & \Sigma(\alpha : \lceil \kappa_1 \rceil^\Gamma)\,\lceil \kappa_2 \rceil^{\Gamma, \alpha :: \lceil \kappa_1 \rceil^\Gamma}
\end{array}
$$

$$
\lceil \alpha \rceil^\Gamma \triangleq \begin{cases} \eta_{\Gamma(\alpha)}\;\alpha & \text{if } \alpha \in \mathrm{dom}\,\Gamma \\ \alpha & \text{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
\lceil \eta_\kappa \rceil^\Gamma & \triangleq & \eta_{\lceil \kappa \rceil^\Gamma} \\
\lceil \lambda(\alpha :: \kappa)\,\tau \rceil^\Gamma & \triangleq & \lambda(\alpha :: \lceil \kappa \rceil^\Gamma)\,\lceil \tau \rceil^{\Gamma, \alpha :: \lceil \kappa \rceil^\Gamma} \\
\lceil \tau_1\;\tau_2 \rceil^\Gamma & \triangleq & \lceil \tau_1 \rceil^\Gamma\;\lceil \tau_2 \rceil^\Gamma \\
\lceil (\tau_1, \tau_2) \rceil^\Gamma & \triangleq & (\lceil \tau_1 \rceil^\Gamma, \lceil \tau_2 \rceil^\Gamma) \\
\lceil \tau.\mathrm{i} \rceil^\Gamma & \triangleq & \lceil \tau \rceil^\Gamma.\mathrm{i}
\end{array}
$$

As in the case of the simply typed λ-calculus, we define erasure of expansors.

Definition 3.5.10 (Erasure of expansors). The erasure of expansors $\lfloor \cdot \rfloor$ is the homomorphism generated by $\lfloor \alpha \rfloor = \alpha$ and $\lfloor \eta_\kappa \rfloor = \lambda(\alpha :: \lfloor \kappa \rfloor)\,\alpha$. The full definition is given below:

$$
\begin{array}{rcl}
\lfloor \alpha \rfloor & \triangleq & \alpha \\
\lfloor \eta_\kappa \rfloor & \triangleq & \lambda(\alpha : \lfloor \kappa \rfloor)\,\alpha \\
\lfloor \lambda(\alpha : \kappa)\,\tau \rfloor & \triangleq & \lambda(\alpha : \lfloor \kappa \rfloor)\,\lfloor \tau \rfloor \\
\lfloor \tau_1\;\tau_2 \rfloor & \triangleq & \lfloor \tau_1 \rfloor\;\lfloor \tau_2 \rfloor \\
\lfloor (\tau_1, \tau_2) \rfloor & \triangleq & (\lfloor \tau_1 \rfloor, \lfloor \tau_2 \rfloor) \\
\lfloor \tau.\ell \rfloor & \triangleq & \lfloor \tau \rfloor.\ell
\end{array}
\qquad
\begin{array}{rcl}
\lfloor \star \rfloor & \triangleq & \star \\
\lfloor \mathcal{S}(\tau) \rfloor & \triangleq & \mathcal{S}(\lfloor \tau \rfloor) \\
\lfloor \Pi(\alpha : \kappa)\,\tau \rfloor & \triangleq & \Pi(\alpha : \lfloor \kappa \rfloor)\,\lfloor \tau \rfloor \\
\lfloor \Sigma(\alpha : \kappa)\,\tau \rfloor & \triangleq & \Sigma(\alpha : \lfloor \kappa \rfloor)\,\lfloor \tau \rfloor
\end{array}
$$

The original system of Stone and Harper does not contain expansors. We could have extended it, but then it would have been necessary to redo its full metatheory. Instead, we define the judgments

of our system with respect to Stone-Harper's judgments up to erasure of expansors.

Definition 3.5.11. The wellformedness judgments of the system with expansors are defined as follows:

- $\Gamma \vdash$ ok holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}}$ ok holds;
- $\Gamma \vdash \kappa$ ok holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}} \lfloor\kappa\rfloor$ ok holds;
- $\Gamma \vdash \tau :: \kappa$ holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}} \lfloor\tau\rfloor :: \lfloor\kappa\rfloor$ holds;
- $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}} \lfloor\kappa_1\rfloor \leqslant \lfloor\kappa_2\rfloor$ holds;
- $\Gamma \vdash \kappa_1 \equiv \kappa_2$ holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}} \lfloor\kappa_1\rfloor \equiv \lfloor\kappa_2\rfloor$ holds;
- $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$ holds iff $\lfloor\Gamma\rfloor \vdash_{\mathsf{HS}} \lfloor\tau_1\rfloor \equiv \lfloor\tau_2\rfloor :: \lfloor\kappa\rfloor$ holds.

By construction, the definition ensures that the insertion of expansors or their erasure preserves wellformedness and equivalence. Moreover, when restricted to expansor-free expressions, the definition coincides with Stone and Harper's. According to this definition, $\Gamma \vdash \eta_\kappa :: \kappa \to \kappa$ holds as long as $\Gamma \vdash \kappa$ ok holds. Furthermore, $\Gamma \vdash \eta_\kappa \equiv \lambda(\alpha :: \kappa)\, \alpha :: \kappa \to \kappa$ is also true. Consequently, it is also true that $\Gamma \vdash \eta_\kappa\ \tau \equiv \tau :: \kappa$, as long as $\Gamma \vdash \tau :: \kappa$.

Lemma 3.5.2. *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \kappa$, *then* $\Gamma \vdash \tau \equiv \lceil\tau\rceil^{\lceil\Gamma\rceil} :: \kappa$;
- *If* $\Gamma \vdash \kappa$ ok, *then* $\Gamma \vdash \kappa \equiv \lceil\kappa\rceil^{\lceil\Gamma\rceil}$.

Proof. By mutual induction on the wellformedness judgments of the Harper-Stone system, using the equivalence between an expansor and the identity. □

3.5.2 Translation into the simply typed λ-calculus

To prove the strong normalization property of $\xrightarrow{\beta\pi}$ on wellformed types and kinds, we first define a translation into the simply typed λ-calculus with pairs and unit, from which we can transfer the strong normalization property. This is a common technique. To succeed, the translation must preserve wellformedness, but also must enjoy a forward simulation result: every reduction step in the source should translate into one or more reduction steps in the target. In particular, the transformation translates kinds into terms, so that any kind annotation on the argument of functions can be inserted in the target: this way, reductions in the kinds are preserved by the translation.

Definition 3.5.12 (Translation into the simply-typed λ-calculus with pairs and unit)**.**

$$
\begin{array}{rcll}
|\star| & \triangleq & (\star, ()) & \\
|\mathcal{S}(\tau)| & \triangleq & (\star, |\tau|) & \\
|\Pi(\alpha : \kappa_1)\, \kappa_2| & \triangleq & (\kappa_1' \to \kappa_2', (\lambda(\beta :: \kappa_1')\, \lambda(\alpha :: \kappa_1')\, \tau_2)\ \tau_1) & \\
 & & & \text{where } \beta \text{ fresh, } |\kappa_1| = (\kappa_1', \tau_1) \text{ and } |\kappa_2| = (\kappa_2', \tau_2) \\
|\Sigma(\alpha : \kappa_1)\, \kappa_2| & \triangleq & (\kappa_1' \times \kappa_2', (\lambda(\alpha :: \kappa_1')\, (\alpha, \tau_2))\ \tau_1) & \text{where } |\kappa_1| = (\kappa_1', \tau_1) \text{ and } |\kappa_2| = (\kappa_2', \tau_2)
\end{array}
$$

$$
\begin{array}{rcll}
|\alpha| & \triangleq & \alpha & \\
|\eta_\kappa| & \triangleq & |\lambda(\alpha :: \kappa)\, \alpha| & \\
|\lambda(\alpha :: \kappa)\, \tau| & \triangleq & (\lambda(\beta :: \kappa_1)\, \lambda(\alpha :: \kappa_1)\ |\tau|)\ \tau_1 & \text{where } \beta \text{ fresh and } |\kappa| = (\kappa_1, \tau_1) \\
|\tau_1\ \tau_2| & \triangleq & |\tau_1|\ |\tau_2| & \\
|(\tau_1, \tau_2)| & \triangleq & (|\tau_1|, |\tau_2|) & \\
|\tau.i| & \triangleq & |\tau|.i &
\end{array}
$$

$$
\begin{array}{rcll}
|\varepsilon| & \triangleq & \varepsilon & \\
|\Gamma, \alpha :: \kappa| & \triangleq & |\Gamma|, \alpha :: \kappa_1 & \text{where } |\kappa| = (\kappa_1, \tau_1)
\end{array}
$$

Proposition 3.5.3 (Invariants of the translation)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$ *and* $|\kappa_i| = (\kappa_i', \tau_i)$*, then* $\kappa_1' = \kappa_2'$*;*
- *If* $\Gamma \vdash \kappa\ \mathsf{ok}$ *and* $|\kappa| = (\kappa', \tau)$*, then* $|\Gamma| \vdash_{\mathsf{ST}} \tau :: \kappa'$*;*
- *If* $\Gamma \vdash \tau :: \kappa$ *and* $|\kappa| = (\kappa', \tau')$*, then* $|\Gamma| \vdash_{\mathsf{ST}} |\tau| :: \kappa'$*;*
- *If* $\tau \xrightarrow{\beta\pi} \tau'$ *then* $|\tau| \xrightarrow{\beta\pi}{}^{+} |\tau'|$
- *If* $\kappa \xrightarrow{\beta\pi} \kappa'$ *and* $|\kappa| = (\kappa_1, \tau_1)$ *and* $|\kappa'| = (\kappa_1', \tau_1')$ *then* $\tau_1 \xrightarrow{\beta\pi}{}^{+} \tau_1'$ *and* $\kappa_1 = \kappa_1'$*.*

Proof. The first item is proved by induction on the subkinding judgment. The next two items are proved by mutual induction on the wellformedness judgments. The last two items are proved by mutual induction on the reduction relation. □

The above proposition states the preservation of types and the desired simulation property between a source and its image through the translation.

3.5.3 Subject reduction

Proposition 3.5.4 (Subject reduction)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \tau :: \kappa$ *and* $\tau \xrightarrow{\beta\pi} \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*;*
- *If* $\Gamma \vdash \tau :: \kappa$ *and* $\tau \xrightarrow{\eta^\bullet} \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*;*
- *If* $\Gamma \vdash \tau :: \kappa$ *and* $\tau \xrightarrow{\beta\pi\eta^\bullet} \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*.*

Proof. The assertions hold for the following reasons:

- The first item follows from the admissibility of β and π in the singleton kinds calculus (Proposition 3.1.8).
- The second item follows from the admissibility of β and η in the singleton kinds calculus (Proposition 3.1.8) and from the fact that the expansion constants are, by definition, equivalent to the identity.

- The third item is the combination of the two previous ones. □

3.5.4 Confluence and strong normalization

As usual, $\xrightarrow{\beta\pi}$ commute with substitution as follows.

Lemma 3.5.5. *The following assertions hold:*

- *If* $\tau_1 \xrightarrow{\beta\pi} \tau_1'$, *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\beta\pi} \tau_1'[\alpha \leftarrow \tau_2]$;
- *If* $\kappa_1 \xrightarrow{\beta\pi} \kappa_1'$, *then* $\kappa_1[\alpha \leftarrow \tau_2] \xrightarrow{\beta\pi} \kappa_1'[\alpha \leftarrow \tau_2]$;
- *If* $\tau_2 \xrightarrow{\beta\pi} \tau_2'$, *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\beta\pi}{}^{\star} \tau_1[\alpha \leftarrow \tau_2']$;
- *If* $\tau_2 \xrightarrow{\beta\pi} \tau_2'$, *then* $\kappa_1[\alpha \leftarrow \tau_2] \xrightarrow{\beta\pi}{}^{\star} \kappa_1[\alpha \leftarrow \tau_2']$.

The relation $\xrightarrow{\eta\bullet}$ enjoys the same properties.

Lemma 3.5.6. *The following assertions hold:*

- *If* $\tau_1 \xrightarrow{\eta\bullet} \tau_1'$, *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet} \tau_1'[\alpha \leftarrow \tau_2]$;
- *If* $\kappa_1 \xrightarrow{\eta\bullet} \kappa_1'$, *then* $\kappa_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet} \kappa_1'[\alpha \leftarrow \tau_2]$;
- *If* $\tau_2 \xrightarrow{\eta\bullet} \tau_2'$, *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet}{}^{\star} \tau_1[\alpha \leftarrow \tau_2']$;
- *If* $\tau_2 \xrightarrow{\eta\bullet} \tau_2'$, *then* $\kappa_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet}{}^{\star} \kappa_1[\alpha \leftarrow \tau_2']$.

The relation $\xrightarrow{\eta\bullet\downarrow}$ enjoys the following properties.

Lemma 3.5.7. *The following assertions hold:*

- *If* $\tau_1 \xrightarrow{\eta\bullet\downarrow} \tau_1'$ *and* $\tau_2 \xrightarrow{\eta\bullet\downarrow} \tau_2'$, *then* $\tau_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet\downarrow} \tau_1'[\alpha \leftarrow \tau_2']$;
- *If* $\kappa_1 \xrightarrow{\eta\bullet\downarrow} \kappa_1'$ *and* $\tau_2 \xrightarrow{\eta\bullet\downarrow} \tau_2'$, *then* $\kappa_1[\alpha \leftarrow \tau_2] \xrightarrow{\eta\bullet\downarrow} \kappa_1'[\alpha \leftarrow \tau_2']$.

We first establish strong normalization and confluence for $\xrightarrow{\beta\pi}$.

Proposition 3.5.8. $\xrightarrow{\beta\pi}$ *is strongly normalizing on wellformed types and kinds.*

Proof. From well-known strong normalization of simply typed λ-calculus and Proposition 3.5.3 and Proposition 3.5.4. □

Proposition 3.5.9. $\xrightarrow{\beta\pi}$ *is weakly confluent.*

Proof. By case analysis, using Lemma 3.5.5. □

Proposition 3.5.10. $\xrightarrow{\beta\pi}$ *is confluent on wellformed types and kinds.*

Proof. Follows from Proposition 3.5.8, Proposition 3.5.9 and Newman's Lemma 3.3.5 on page 66. □

Now, we prove strong normalization and confluence for the relation $\xrightarrow{\eta\bullet}$.

Proposition 3.5.11. $\xrightarrow{\eta\bullet}$ *is strongly normalizing.*

Proof. This is proved using the multiset order of size of kinds of expansors, that is well-founded, making use of Lemma 3.5.1 on page 77. □

Proposition 3.5.12. $\xrightarrow{\eta^\bullet}$ *is weakly confluent.*

Proof. By case analysis, using Lemma 3.5.6. □

Proposition 3.5.13. $\xrightarrow{\eta^\bullet}$ *is confluent.*

Proof. Follows from Proposition 3.5.12, Proposition 3.5.11 on the previous page and Newman's Lemma 3.3.5 on page 66. □

In the same way as in Section 3.4.3 on page 70, we establish confluence and strong normalization for the full reduction relation, considered on wellformed types and kinds.

Lemma 3.5.14. *The relation* $\xrightarrow{\beta\pi}$ DPG*-commutes with* $\xrightarrow{\eta^\bullet}$.

Proof. By induction on $\xrightarrow{\beta\pi}$ and case analysis on $\xrightarrow{\eta^\bullet}$, using Lemma 3.5.6 on the preceding page. □

Theorem 3.5.15 (Confluence)**.** $\xrightarrow{\beta\pi\eta^\bullet}$ *is confluent on wellformed types and kinds.*

Proof. By Lemma 3.3.11 on page 67, using Lemma 3.5.14, Proposition 3.5.10, Proposition 3.5.13, Proposition 3.5.8 and Proposition 3.5.4. □

Lemma 3.5.16. $\xrightarrow{\beta\pi}$ *preserves* $\xrightarrow{\eta^\bullet}$*-normal forms.*

Proof. By induction on $\xrightarrow{\beta\pi}$ and using lemma 3.5.7 on the preceding page. □

Theorem 3.5.17 (Strong normalization)**.** $\xrightarrow{\beta\pi\eta^\bullet}$ *is strongly normalizing on wellformed types and kinds.*

Proof. By Lemma 3.3.12 on page 67, using Lemma 3.5.14, Proposition 3.5.8, Proposition 3.5.11 and Lemma 3.5.16. □

3.5.5 Properties of expansors

Lemma 3.5.18. *For all* Γ, Γ', α, β, κ, κ', τ, *if* $\alpha, \beta \notin \operatorname{dom}\Gamma, \operatorname{dom}\Gamma'$ *then:*

- $\lceil\tau\rceil^{\Gamma,\alpha::\kappa',\Gamma'}[\alpha \leftarrow \beta] = \lceil\tau[\alpha \leftarrow \beta]\rceil^{\Gamma,\beta::\kappa',\Gamma'[\alpha\leftarrow\beta]}$
- $\lceil\kappa\rceil^{\Gamma,\alpha::\kappa',\Gamma'}[\alpha \leftarrow \beta] = \lceil\kappa[\alpha \leftarrow \beta]\rceil^{\Gamma,\beta::\kappa',\Gamma'[\alpha\leftarrow\beta]}$

Proof. By mutual induction on τ and κ. □

As shown for the simply typed λ-calculus, expansors are idempotent in the singleton kind calculus.

Lemma 3.5.19 (Idempotency of expansors)**.** *For all* τ *and* κ, *we have* $\eta_\kappa\ (\eta_\kappa\ \tau) \xleftrightarrow{\beta\pi\eta^\bullet} \eta_\kappa\ \tau$.

Proof. By induction on the size of κ, using Lemma 3.5.1 on page 77.

- $\underline{\kappa = \star}$: immediate.
- $\underline{\kappa = \mathcal{S}(\tau')}$: immediate.

- $\underline{\kappa = \Pi(\alpha : \kappa_1)\,\kappa_2}$:

$$
\begin{array}{ll}
 & \eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ (\eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ \tau) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ \tau\ (\eta_{\kappa_1}\ \alpha)) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ ((\lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\tau\ (\eta_{\kappa_1}\ \alpha)))\ (\eta_{\kappa_1}\ \alpha)) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\eta_{\kappa_1}\ \alpha)]}\ (\tau\ (\eta_{\kappa_1}\ (\eta_{\kappa_1}\ \alpha)))) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\tau\ (\eta_{\kappa_1}\ \alpha))) & (\mathbf{1}) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \kappa_1)\,\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\tau\ (\eta_{\kappa_1}\ \alpha)) & (\mathbf{2}) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ \tau
\end{array}
$$

  since, by induction hypothesis:
  (1) $\eta_{\kappa_1}\ (\eta_{\kappa_1}\ \alpha) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_{\kappa_1}\ \alpha$ and
  (2) $\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\tau\ (\eta_{\kappa_1}\ \alpha))) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \alpha]}\ (\tau\ (\eta_{\kappa_1}\ \alpha))$.

- $\underline{\kappa = \Sigma(\alpha : \kappa_1)\,\kappa_2}$:

$$
\begin{array}{ll}
 & \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ (\eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ (\eta_{\kappa_1}\ (\tau.1), \eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2)) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\kappa_1}\ (\eta_{\kappa_1}\ (\tau.1)), \eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\eta_{\kappa_1}\ (\tau.1))]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2))) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\kappa_1}\ \tau.1, \eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \tau.1]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2))) & (\mathbf{1}) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\kappa_1}\ \tau.1, \eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2)) & (\mathbf{2}) \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau
\end{array}
$$

  since, by induction hypothesis:
  (1) $\eta_{\kappa_1}\ (\eta_{\kappa_1}\ (\tau.1)) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \eta_{\kappa_1}\ (\tau.1)$ and
  (2) $\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2)) \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} (\eta_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ (\tau.1)]}\ (\tau.2))$ □

Lemma 3.5.20. *Assume* $\alpha \notin \mathrm{fv}(\Gamma) \cup \mathrm{dom}\,\Gamma' \cup \mathrm{fv}(\kappa)$. *The following assertions hold:*

- $\lceil\tau\rceil^{\Gamma,\alpha::\kappa',\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\tau\rceil^{\Gamma,\alpha::\kappa',\lceil\Gamma'\rceil}[\alpha \leftarrow \eta_{\kappa'}\ \alpha]$;
- $\lceil\kappa\rceil^{\Gamma,\alpha::\kappa'} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\kappa\rceil^{\Gamma,\alpha::\kappa'}[\alpha \leftarrow \eta_{\kappa'}\ \alpha]$.

Proof. By mutual induction, using Lemma 3.5.19. □

As in Section 3.4, erasure is idempotent and commutes with substitution.

Lemma 3.5.21. *The following assertions hold:*

- $\lfloor\lfloor\tau\rfloor\rfloor = \lfloor\tau\rfloor$ *and* $\lfloor\lfloor\kappa\rfloor\rfloor = \lfloor\kappa\rfloor$;
- $\lfloor\tau_1[\alpha \leftarrow \tau_2]\rfloor = \lfloor\tau_1\rfloor[\alpha \leftarrow \lfloor\tau_2\rfloor]$ *and* $\lfloor\kappa[\alpha \leftarrow \tau_2]\rfloor = \lfloor\kappa\rfloor[\alpha \leftarrow \lfloor\tau_2\rfloor]$.

Similarly to Section 3.4, insertion of expansors enjoys the following results about its commutation with substitution.

Lemma 3.5.22. *If* $\mathrm{dom}\,\Gamma' \cap (\mathrm{fv}(\kappa_2) \cup \mathrm{fv}(\tau_2) \cup \mathrm{dom}\,\Gamma \cup \mathrm{fv}(\Gamma) \cup \{\alpha\}) = \emptyset$ *and* $\alpha \notin \mathrm{dom}\,\Gamma \cup \mathrm{fv}(\Gamma)$, *then the following assertions hold:*

- $\lceil\tau_1\rceil^{\Gamma,\alpha::\kappa_2,\Gamma'}[\alpha \leftarrow \lceil\tau_2\rceil^{\Gamma,\Gamma'}] = \lceil\tau_1[\alpha \leftarrow \eta_{\kappa_2}\ \tau_2]\rceil^{\Gamma,\Gamma'[\alpha\leftarrow\lceil\tau_2\rceil^{\Gamma,\Gamma'}]}$

- $\lceil \kappa_1 \rceil^{\Gamma,\alpha::\kappa_2,\Gamma'}[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}] = \lceil \kappa_1[\alpha \leftarrow \eta_{\kappa_2}\ \tau_2] \rceil^{\Gamma,\Gamma'[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}]}$
- $\lceil \tau_1 \rceil^{\Gamma,\Gamma'}[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}] = \lceil \tau_1[\alpha \leftarrow \tau_2] \rceil^{\Gamma,\Gamma'[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}]}$
- $\lceil \kappa_1 \rceil^{\Gamma,\Gamma'}[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}] = \lceil \kappa_1[\alpha \leftarrow \tau_2] \rceil^{\Gamma,\Gamma'[\alpha \leftarrow \lceil \tau_2 \rceil^{\Gamma,\Gamma'}]}$

3.5.6 Soundness of convertibility

The soundness property of convertibility with respect to the equivalence judgments is heavily based on the results of Proposition 3.1.8 on page 61.

Lemma 3.5.23. *The following assertions hold:*

- *If* $\Gamma \vdash \tau_1 :: \kappa$ *and* $\tau_1 \xrightarrow{\beta\pi\eta\bullet} \tau_2$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$ *holds;*
- *If* $\Gamma \vdash \kappa_1\ \mathsf{ok}$ *and* $\kappa_1 \xrightarrow{\beta\pi\eta\bullet} \kappa_2$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_2$ *holds.*

Proof. By induction on the reduction relation, using Lemma 3.5.21. The case of β and π are handled by the admissibility of β and π (see Proposition 3.1.8 on page 61). The case of $\eta_\bullet$ is handled by the admissibility of η (see Proposition 3.1.8) and the fact that $\Gamma \vdash \eta_\kappa \equiv \lambda(\alpha : \kappa)\ \alpha :: \kappa \to \kappa$. The correctness of contextual closure of the reduction follows from the contextual rules for the equivalence. □

The above lemma on reduction generalizes to convertibility as follows.

Proposition 3.5.24. *The following assertions hold:*

- *If* $\Gamma \vdash \tau_1 :: \kappa$ *and* $\Gamma \vdash \tau_2 :: \kappa$ *and* $\tau_1 \xleftrightarrow{\beta\pi\eta\bullet} \tau_2$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$ *holds;*
- *If* $\Gamma \vdash \kappa_1\ \mathsf{ok}$ *and* $\Gamma \vdash \kappa_2\ \mathsf{ok}$ *and* $\kappa_1 \xleftrightarrow{\beta\pi\eta\bullet} \kappa_2$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_2$ *holds.*

Proof. By induction on convertibility, using Lemma 3.5.23 and Proposition 3.5.4. □

We can finally state the soundness theorem. It differs from Proposition 3.5.24 in the sense that it inserts expansors in the types and kinds under consideration, so that it is the converse of the completeness theorem (Theorem 3.5.40 on page 98). Inserting expansors is not important for equivalence.

Theorem 3.5.25 (Soundness)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \tau_1 :: \kappa$ *and* $\Gamma \vdash \tau_2 :: \kappa$ *and* $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil\Gamma\rceil} \xleftrightarrow{\beta\pi\eta\bullet} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil\Gamma\rceil}$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa$;
- *If* $\Gamma \vdash \kappa_1\ \mathsf{ok}$ *and* $\Gamma \vdash \kappa_2\ \mathsf{ok}$ *and* $\lceil \kappa_1 \rceil^{\lceil\Gamma\rceil} \xleftrightarrow{\beta\pi\eta\bullet} \lceil \kappa_2 \rceil^{\lceil\Gamma\rceil}$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_2$;

Proof. By Lemma 3.5.2 on page 79 and Proposition 3.5.24. □

Because the Stone-Harper judgments coincide with ours when they are restricted to expansor-free types and kinds, we can restate the theorem as follows.

Theorem 3.5.26 (Soundness)**.** *The following assertions hold:*

- *If* $\Gamma \vdash_{\mathsf{HS}} \tau_1 :: \kappa$ *and* $\Gamma \vdash \tau_2 :: \kappa$ *and* $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil\Gamma\rceil} \xleftrightarrow{\beta\pi\eta\bullet} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil\Gamma\rceil}$, *then* $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa$;
- *If* $\Gamma \vdash_{\mathsf{HS}} \kappa_1\ \mathsf{ok}$ *and* $\Gamma \vdash_{\mathsf{HS}} \kappa_2\ \mathsf{ok}$ *and* $\lceil \kappa_1 \rceil^{\lceil\Gamma\rceil} \xleftrightarrow{\beta\pi\eta\bullet} \lceil \kappa_2 \rceil^{\lceil\Gamma\rceil}$, *then* $\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_2$;

3.5.7 Completeness of convertibility

The completeness theorem is stated as follows, for the case of types:

$$\text{If } \Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa \text{ holds, then } \lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil \Gamma \rceil}.$$

It reduces equivalence to a convertibility test between two types after expansors have been inserted. A direct proof by mutual induction on the wellformedness, subkinding and equivalence judgments does not go through.

For instance, in the case of the rule EqApp, we have as hypotheses that $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_1' :: \Pi(\alpha : \kappa_2)\,\kappa_1$ and $\Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv \tau_2' :: \kappa_2$. By induction hypotheses, one gets $\lceil \eta_{\Pi(\alpha:\kappa_2)\,\kappa_1}\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\Pi(\alpha:\kappa_2)\,\kappa_1}\ \tau_1' \rceil^{\lceil \Gamma \rceil}$ and $\lceil \eta_{\kappa_2}\ \tau_2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_2}\ \tau_2' \rceil^{\lceil \Gamma \rceil}$. From this, one can show

$$\lceil \eta_{\kappa_1[\alpha \leftarrow \eta_{\kappa_2}\ \tau_2]}\ (\tau_1\ (\eta_{\kappa_2}\ \tau_2)) \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_1[\alpha \leftarrow \eta_{\kappa_2}\ \tau_2]}\ (\tau_1'\ (\eta_{\kappa_2}\ \tau_2')) \rceil^{\lceil \Gamma \rceil}$$

whereas one would like to show the same statement, without the expansors η_{κ_2}:

$$\lceil \eta_{\kappa_1[\alpha \leftarrow \tau_2]}\ (\tau_1\ \tau_2) \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_1[\alpha \leftarrow \tau_2]}\ (\tau_1'\ \tau_2') \rceil^{\lceil \Gamma \rceil}$$

As a consequence, the direct proof gets stuck. Stone and Harper encountered a similar difficulty when trying to prove completeness for their normalization algorithm, because normalization of application is not defined using the normalized parts of an application. In our case, however, this is true, but as we just showed, the external expansion is problematic. After several unsuccessful attempts to generalize the completeness property, we eventually used the same technique as Stone and Harper, based an a Kripke logical relation. We describe it now.

Notation. In the following, $\mathcal{G}$ (respectively $\mathcal{T}$, $\mathcal{K}$, and $\mathfrak{S}$) denotes non empty sets of environments (respectively types, kinds, and mappings from variables to sets of types).

Notation. We reuse the same notations as in [SH06] to describe operations on finite sets and pattern matching on them.

$$\begin{array}{rcl}
\mathcal{K}[\alpha \leftarrow \mathcal{T}] & \triangleq & \{\kappa[\alpha \leftarrow \tau] \mid \kappa \in \mathcal{K}, \tau \in \mathcal{T}\} \\
\mathcal{T}_1\ \mathcal{T}_2 & \triangleq & \{\tau_1\ \tau_2 \mid \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2\} \\
\mathcal{T}.\mathrm{i} & \triangleq & \{\tau.\mathrm{i} \mid \tau \in \mathcal{T}\} \\
\eta_{\mathcal{K}} & \triangleq & \{\eta_\kappa \mid \kappa \in \mathcal{K}\} \\
\mathcal{S}(\mathcal{T}) & \triangleq & \{\mathcal{S}(\tau) \mid \tau \in \mathcal{T}\} \\
\Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 & \triangleq & \{\Pi(\alpha : \kappa_1)\,\kappa_2 \mid \kappa_1 \in \mathcal{K}_1, \kappa_2 \in \mathcal{K}_2\} \\
\Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 & \triangleq & \{\Sigma(\alpha : \kappa_1)\,\kappa_2 \mid \kappa_1 \in \mathcal{K}_1, \kappa_2 \in \mathcal{K}_2\} \\
\mathfrak{S}(\mathcal{T}) & \triangleq & \{\sigma(\tau) \mid \sigma \in \mathfrak{S}, \tau \in \mathcal{T}\} \\
\mathfrak{S}(\mathcal{K}) & \triangleq & \{\sigma(\kappa) \mid \sigma \in \mathfrak{S}, \kappa \in \mathcal{K}\} \\
\mathfrak{S}; \alpha \mapsto \mathcal{T} & \triangleq & \{\sigma; \alpha \mapsto \tau \mid \sigma \in \mathfrak{S}, \tau \in \mathcal{T}\}
\end{array}$$

Definition 3.5.13 (Inclusion of environments)**.** We say that Γ is *included* in Γ', denoted by $\Gamma \subseteq \Gamma'$ iff for every $x \in \operatorname{dom}\Gamma$, then $x \in \operatorname{dom}\Gamma'$ and $\Gamma(x) = \Gamma'(x)$ and $(\operatorname{dom}\Gamma' \setminus \operatorname{dom}\Gamma) \cap \mathrm{fv}(\Gamma) = \emptyset$.

Definition 3.5.14 (Later environment set)**.** We say that $\mathcal{G}'$ is *later* than $\mathcal{G}$, denoted by $\mathcal{G}' \sqsupseteq \mathcal{G}$, if for all $\Gamma' \in \mathcal{G}'$, there exists $\Gamma \in \mathcal{G}$ such that $\Gamma \subseteq \Gamma'$.

Definition 3.5.15 (Logical relation)**.** The logical relations on types and kinds are defined as follows:

- $\mathcal{G} \models \mathcal{K}$ holds if:

  - $\mathcal{K} = \{\star\}$
  - or, $\mathcal{K} = \mathcal{S}(\mathcal{U})$ and $\mathcal{G} \models \mathcal{U} :: \{\star\}$ holds;
  - or, $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\mathcal{G} \models \mathcal{K}_1$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;
  - or, $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\mathcal{G} \models \mathcal{K}_1$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;

- $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ holds if $\mathcal{G} \models \mathcal{K}$ holds and:
  - $\mathcal{K} = \{\star\}$ and for every $\Gamma \in \mathcal{G}$ and $\tau_1, \tau_2 \in \mathcal{T}$, $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$;
  - or, $\mathcal{K} = \mathcal{S}(\mathcal{U})$ and $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$
  - or, $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;
  - or, $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ and $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$.

The logical relation is meant to be understood as a relation between subsets of pseudo-equivalence classes.

The logical relation on kinds relates sets of kinds in some sets of environments, which is denoted by $\mathcal{G} \models \mathcal{K}$. The base kind $\star$ is related to itself only. Singleton kinds relate types that are already related at kind $\star$. The case of arrow and pair kinds is usual for logical relations: the left parts must be related, while the right parts must be related up to any substitution with types that are themselves related in any later environment.

The logical relation on types relates sets of types to sets of kinds in some sets of environments. Types are related at the base kind if they are all pairwise convertible up to insertion of annotations. This is the only place where our definition differs from [SH06]: they required that there exists a common normal form, *i.e.* a unique output of the normalization procedure, for any type and any environment in the given sets. Types are related at singleton kinds when they are also related at kind $\star$ to the types from the singletons. This is reminiscent of the fact that a type τ has kind $\mathcal{S}(\mathfrak{u})$ iff τ and $\mathfrak{u}$ are equivalent. The cases for arrow and pair kinds are usual: they use an extensional style. More specifically, types are related at arrow types, if their applications to any argument that are related in any later environment are themselves related. Similarly, types are related at pair kinds if their first projections (and *resp.* their second projections) are related.

Definition 3.5.16 (Logically valid set of substitution)**.** $\mathcal{G} \models \mathfrak{S} : \Gamma$ holds if for all $\alpha \in \operatorname{dom}\Gamma$, $\mathcal{G} \models \mathfrak{S}(\alpha) :: \mathfrak{S}(\Gamma(\alpha))$.

We follow the same proof strategy as Stone and Harper: we use the logical relation as an intermediate invariant, that is strong enough to prove the completeness property. More precisely, we proceed in two stages:

- first, we show (Proposition 3.5.35 on page 91) that the logical relation entails the desired property on convertibility: if $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$, then for every $\Gamma \in \mathcal{G}$, every $\tau_1, \tau_2 \in \mathcal{T}$ and every $\kappa \in \mathcal{K}$, $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds (and similarly for kinds);
- Then, we show (Proposition 3.5.39 on page 98) that the logical relation is a consequence of the judgments: if $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa$, then $\{\Gamma\} \models \{\tau_1, \tau_2\} :: \{\kappa\}$ holds (and similar results for the other judgments).

The following lemma is necessary to allow a later induction reasoning on the sizes of kinds in the logical relation.

Lemma 3.5.27 (Uniqueness of sizes of kinds)**.** *The following assertions hold:*

- *If* $\mathcal{G} \models \mathcal{K}$, *then for every* $\kappa_1, \kappa_2 \in \mathcal{K}$, $\operatorname{size} \kappa_1 = \operatorname{size} \kappa_2$;
- *If* $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$, *then for every* $\kappa_1, \kappa_2 \in \mathcal{K}$, $\operatorname{size} \kappa_1 = \operatorname{size} \kappa_2$;

In this case, we write $\operatorname{size} \mathcal{K}$ *to denote the size of any element of* $\mathcal{K}$, *since* $\mathcal{K}$ *is by definition non-empty.*

Proof. By induction on the definition of the logical relation. □

We need to show that convertibility up to insertion enjoys a weakening property, which needs a intermediate lemma on the insertion function.

Lemma 3.5.28. *Assume* $\Gamma \subseteq \Gamma'$. *Let* σ *be the substitution of domain* $\operatorname{dom}\Gamma' \setminus \operatorname{dom}\Gamma$ *such that for every* $\alpha \in \operatorname{dom}\Gamma' \setminus \operatorname{dom}\Gamma$, $\sigma(\alpha) \triangleq \eta_{\Gamma'(\alpha)}\ \alpha$. *Then for every type* τ *and kind* κ, $\lceil\tau\rceil^{\Gamma'} = \lceil\tau\rceil^{\Gamma}\sigma$ *and* $\lceil\kappa\rceil^{\Gamma'} = \lceil\kappa\rceil^{\Gamma}\sigma$.

Proof. By mutual structural induction on types and kinds. □

Lemma 3.5.29. *Assume* $\Gamma \subseteq \Gamma'$. *The following assertions hold:*

- *if* $\lceil\tau_1\rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\tau_2\rceil^{\Gamma}$, *then* $\lceil\tau_1\rceil^{\Gamma'} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\tau_2\rceil^{\Gamma'}$ *holds;*
- *if* $\lceil\kappa_1\rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\kappa_2\rceil^{\Gamma}$, *then* $\lceil\kappa_1\rceil^{\Gamma'} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil\kappa_2\rceil^{\Gamma'}$ *holds.*

Proof. Follows from Lemma 3.5.28 and stability of convertibility under substitution. □

The next lemma states that the logical relation is monotone, *i.e.* it is preserved by taking later environments.

Lemma 3.5.30 (Monotonicity)**.** *The following assertions hold:*

- *If* $\mathcal{G} \models \mathcal{K}$ *and* $\mathcal{G}' \sqsupseteq \mathcal{G}$, *then* $\mathcal{G}' \models \mathcal{K}$;
- *If* $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ *and* $\mathcal{G}' \sqsupseteq \mathcal{G}$, *then* $\mathcal{G}' \models \mathcal{T} :: \mathcal{K}$;

Proof. By induction on the sizes of sets of kinds. The base cases follow from Lemma 3.5.29. □

The next lemmas show that the sets in the relation are subsets of pseudo-equivalence classes. We begin by showing that the relation is stable by taking non-empty subsets and by taking overlapping unions.

Lemma 3.5.31. *The following assertions hold:*

(a) If $\mathcal{G} \models \mathcal{K}$ *and* $\mathcal{K}' \subseteq \mathcal{K}$, *then* $\mathcal{G} \models \mathcal{K}'$;

(b) If $\mathcal{G} \models \mathcal{K}$ *and* $\mathcal{G} \models \mathcal{K}'$ *and* $\mathcal{K} \cap \mathcal{K}' \neq \emptyset$ *then* $\mathcal{G} \models \mathcal{K} \cup \mathcal{K}'$;

(c) If $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ *and* $\mathcal{K}' \subseteq \mathcal{K}$ *and* $\mathcal{T}' \subseteq \mathcal{T}$, *then* $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}'$;

(d) If $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ *and* $\mathcal{G} \models \mathcal{K}'$ *and* $\mathcal{K} \cap \mathcal{K}' \neq \emptyset$ *then* $\mathcal{G} \models \mathcal{T} :: \mathcal{K} \cup \mathcal{K}'$;

(e) If $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ *and* $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}$ *and* $\mathcal{T} \cap \mathcal{T}' \neq \emptyset$ *then* $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \mathcal{K}$.

Proof. By induction on the size of sets of kinds (everywhere but on the base case, the proof is identical to the one of [SH06]):

- $\mathcal{K} = \{\star\}$:
  (a) Since $\emptyset \subsetneq \mathcal{K}' \subseteq \{\star\}$, we have $\mathcal{K}' = \{\star\}$. Then $\mathcal{G} \models \mathcal{K}'$ holds by assumption.

(b) Since $\{\star\} \cap \mathcal{K}' \neq \emptyset$, we have $\{\star\} \subseteq \mathcal{K}'$ and then $\{\star\} \cup \mathcal{K}' = \mathcal{K}'$. Then $\mathcal{G} \models \{\star\} \cup \mathcal{K}'$ holds by assumption.

(c) Since $\emptyset \subsetneq \mathcal{K}' \subseteq \{\star\}$, we have $\mathcal{K}' = \{\star\}$. By definition of the logical relation, for all $\Gamma \in \mathcal{G}$ and $\tau_1, \tau_2 \in \mathcal{T}$, $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds. Since $\mathcal{T}' \subseteq \mathcal{T}$, it is also true for all $\tau_1, \tau_2 \in \mathcal{T}'$. Hence $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}'$.

(d) Since $\{\star\} \cap \mathcal{K}' \neq \emptyset$, we have $\{\star\} \subseteq \mathcal{K}'$. By case analysis on $\mathcal{G} \models \mathcal{K}'$, we necessary have $\mathcal{K}' = \{\star\}$. Hence $\mathcal{K} \cup \mathcal{K}' = \{\star\}$. Then $\mathcal{G} \models \mathcal{T} :: \mathcal{K} \cup \mathcal{K}'$ holds by assumption.

(e) Let $\Gamma \in \mathcal{G}$ and $\tau_1, \tau_2 \in \mathcal{T} \cup \mathcal{T}'$. There are four cases:

- $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$: then $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds from $\mathcal{G} \models \mathcal{T} :: \{\star\}$.
- $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}'$: let $\tau_3 \in \mathcal{T} \cap \mathcal{T}'$. We have $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$ because of $\mathcal{G} \models \mathcal{T} :: \{\star\}$. Similarly, we have $\lceil \tau_3 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$ because of $\mathcal{G} \models \mathcal{T}' :: \{\star\}$. Then $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$ holds by transitivity.
- $\tau_1 \in \mathcal{T}'$ and $\tau_2 \in \mathcal{T}$: similar to the previous case.
- $\tau_1 \in \mathcal{T}'$ and $\tau_2 \in \mathcal{T}'$: then $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds from $\mathcal{G} \models \mathcal{T}' :: \{\star\}$.

Then $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds. As a consequence, we have $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \mathcal{K}$.

- $\mathcal{K} = \mathcal{S}(\mathcal{U})$:

(a) We have $\mathcal{G} \models \mathcal{U} :: \{\star\}$. Since $\mathcal{K}' \subseteq \mathcal{S}(\mathcal{U})$, $\mathcal{K}' = \mathcal{S}(\mathcal{U}')$ where $\mathcal{U}' \subseteq \mathcal{U}$. Then by induction hypothesis (c) we have $\mathcal{G} \models \mathcal{U}' :: \{\star\}$. Then $\mathcal{G} \models \mathcal{S}(\mathcal{U}')$, and so $\mathcal{G} \models \mathcal{K}'$ holds.

(b) We have $\mathcal{G} \models \mathcal{U} :: \{\star\}$. Since $\mathcal{S}(\mathcal{U}) \cap \mathcal{K}' \neq \emptyset$, $\mathcal{K}' = \mathcal{S}(\mathcal{U}')$ where $\mathcal{U} \cap \mathcal{U}' \neq \emptyset$. Then, by definition of the logical relation, $\mathcal{G} \models \mathcal{U}' :: \{\star\}$ holds. Then by induction hypothesis (e) we get $\mathcal{G} \models \mathcal{U} \cup \mathcal{U}' :: \{\star\}$. Then $\mathcal{G} \models \mathcal{S}(\mathcal{U} \cup \mathcal{U}')$ holds, and so $\mathcal{G} \models \mathcal{K} \cup \mathcal{K}'$ holds.

(c) We have $\mathcal{G} \models \mathcal{U} :: \{\star\}$ and $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$. Since $\mathcal{K}' \subseteq \mathcal{S}(\mathcal{U})$, $\mathcal{K}' = \mathcal{S}(\mathcal{U}')$ where $\mathcal{U}' \subseteq \mathcal{U}$. Then $\mathcal{T}' \cup \mathcal{U}' \subseteq \mathcal{T} \cup \mathcal{U}$. Then by induction hypothesis (c) we have $\mathcal{G} \models \mathcal{U}' :: \{\star\}$ and $\mathcal{G} \models \mathcal{T}' \cup \mathcal{U}' :: \{\star\}$. Then by definition of the logical relation, we have $\mathcal{G} \models \mathcal{S}(\mathcal{U}')$ and $\mathcal{G} \models \mathcal{T}' :: \mathcal{S}(\mathcal{U}')$ and so $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}'$.

(d) We have $\mathcal{G} \models \mathcal{U} :: \{\star\}$ and $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$. Since $\mathcal{S}(\mathcal{U}) \cap \mathcal{K}' \neq \emptyset$, $\mathcal{K}' = \mathcal{S}(\mathcal{U}')$ where $\mathcal{U} \cap \mathcal{U}' \neq \emptyset$. Then by definition of the logical relation we have $\mathcal{G} \models \mathcal{U}' :: \star$. Then by induction hypothesis (e) we get $\mathcal{G} \models \mathcal{U} \cup \mathcal{U}' :: \{\star\}$, hence $\Gamma \models \mathcal{S}(\mathcal{U} \cup \mathcal{U}')$ holds. Then, since $(\mathcal{T} \cup \mathcal{U}) \cap \mathcal{U}' \neq \emptyset$, we obtain from induction hypothesis (e) that $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} \cup \mathcal{U}' :: \{\star\}$ holds. Hence $\mathcal{G} \models \mathcal{T} :: \mathcal{S}(\mathcal{U} \cup \mathcal{U}')$, and so $\mathcal{G} \models \mathcal{T} :: \mathcal{K} \cup \mathcal{K}'$.

(e) We have $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$ and $\mathcal{G} \models \mathcal{T}' \cup \mathcal{U} :: \{\star\}$. Since $\mathcal{T} \cap \mathcal{T}' \neq \emptyset$, $(\mathcal{T} \cup \mathcal{U}) \cap (\mathcal{T}' \cup \mathcal{U}) \neq \emptyset$ also holds. Then, by induction hypothesis (e) we get $\mathcal{G} \models (\mathcal{T} \cup \mathcal{U}) \cap (\mathcal{T}' \cup \mathcal{U}) :: \{\star\}$, and then $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \mathcal{S}(\mathcal{U})$ holds.

- $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$:

(a) Since $\mathcal{K}' \subseteq \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$, $\mathcal{K}' = \Pi(\alpha : \mathcal{K}_1')\, \mathcal{K}_2'$ where $\mathcal{K}_1' \subseteq \mathcal{K}_1$ and $\mathcal{K}_2' \subseteq \mathcal{K}_2$. By induction hypothesis (a) we get $\mathcal{G} \models \mathcal{K}_1'$. Now let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T} :: \mathcal{K}_1'$. Then, by Lemma 3.5.30 on the preceding page and induction hypothesis (d) we have $\mathcal{G}' \models \mathcal{T} :: \mathcal{K}_1$. By definition of the logical relation, we then have $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}]$. Since $\mathcal{K}_2'[\alpha \leftarrow \mathcal{T}] \subseteq \mathcal{K}_2[\alpha \leftarrow \mathcal{T}]$, we get by induction hypothesis (a) $\mathcal{G}' \models \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}]$. Then $\mathcal{G} \models \Pi(\alpha : \mathcal{K}_1')\, \mathcal{K}_2'$ holds.

(b) Since $\mathcal{K}' \cap \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2 \neq \emptyset$, $\mathcal{K}' = \Pi(\alpha : \mathcal{K}_1')\, \mathcal{K}_2'$ where $\mathcal{K}_1' \cap \mathcal{K}_1 \neq \emptyset$ and $\mathcal{K}_2' \cap \mathcal{K}_2 \neq \emptyset$. Then, by induction hypothesis (b), $\mathcal{G} \models \mathcal{K}_1 \cup \mathcal{K}_1'$ holds. Now let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume

$\mathcal{G}' \models \mathcal{T} :: \mathcal{K}_1 \cup \mathcal{K}_1'$. By Lemma 3.5.30 on page 87 and induction hypothesis (c) we have $\mathcal{G}' \models \mathcal{T} :: \mathcal{K}_1$ and $\mathcal{G}' \models \mathcal{T} :: \mathcal{K}_1'$. Hence $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}]$ and $\mathcal{G}' \models \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}]$ hold by definition of the logical relation. And since $\mathcal{K}_2[\alpha \leftarrow \mathcal{T}] \cap \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}] \neq \emptyset$, we get by induction hypothesis (b) $\mathcal{G}' \models (\mathcal{K}_2 \cup \mathcal{K}_2')[\alpha \leftarrow \mathcal{T}]$. Hence $\mathcal{G} \models \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \cup \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ holds.

(c) Since $\mathcal{K}' \subseteq \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$, $\mathcal{K}' = \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ where $\mathcal{K}_1' \subseteq \mathcal{K}_1$ and $\mathcal{K}_2' \subseteq \mathcal{K}_2$. As shown in (a) we have $\mathcal{G} \models \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$. Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1'$. Then by Lemma 3.5.30 on page 87 and induction hypothesis (d) we have $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$. Then, by definition of the logical relation, $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$ holds. Then, by induction hypothesis (c), $\mathcal{G}' \models \mathcal{T}'\,\mathcal{T}_1 :: \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}_1]$ holds. Hence $\mathcal{G} \models \mathcal{T}' :: \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$, and so $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}'$ holds.

(d) Since $\mathcal{K}' \cap \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \neq \emptyset$, $\mathcal{K}' = \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ where $\mathcal{K}_1' \cap \mathcal{K}_1 \neq \emptyset$ and $\mathcal{K}_2' \cap \mathcal{K}_2 \neq \emptyset$. As shown in (b) we have $\mathcal{G} \models \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \cup \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$. Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1 \cup \mathcal{K}_1'$. By induction hypothesis (c), $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$ and $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1'$ hold. Then, by definition of the logical relation, $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$ holds. $\mathcal{G}' \models \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}_1]$ also holds by Lemma 3.5.30 on page 87 and the definition of the logical relation. Then, by induction hypothesis (d), we have $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 :: (\mathcal{K}_2 \cup \mathcal{K}_2')[\alpha \leftarrow \mathcal{T}_1]$. Hence $\mathcal{G} \models \mathcal{T} :: \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \cup \Pi(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$, and so $\mathcal{G} \models \mathcal{T} :: \mathcal{K} \cup \mathcal{K}'$ holds.

(e) Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$. Then by definition of the logical relation, $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{K}_1]$ and $\mathcal{G}' \models \mathcal{T}'\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{K}_1]$. Then, since $\mathcal{T}\,\mathcal{T}_1 \cap \mathcal{T}'\,\mathcal{T}_1 \neq \emptyset$, by induction hypothesis (e) we get $\mathcal{G}' \models \mathcal{T}\,\mathcal{T}_1 \cup \mathcal{T}'\,\mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{K}_1]$. Hence, by definition of the logical relation, $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \mathcal{K}_2$ holds.

- $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$:

(a) The proof is identical to the item (a) for case $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$.

(b) The proof is identical to the item (b) for case $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$.

(c) We have $\mathcal{G} \models \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ and $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$. Since $\mathcal{K}' \subseteq \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$, $\mathcal{K}' = \Sigma(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ where $\mathcal{K}_1' \subseteq \mathcal{K}_1$ and $\mathcal{K}_2' \subseteq \mathcal{K}_2$. Then, since $\mathcal{T}'.1 \subseteq \mathcal{T}.1$ we get by induction hypothesis (c) $\mathcal{G} \models \mathcal{T}'.1 :: \mathcal{K}_1'$. Also, since $\mathcal{T}'.1 \subseteq \mathcal{T}.1$ and $\mathcal{K}_2'[\alpha \leftarrow \mathcal{T}'.1] \subseteq \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$ we get by induction hypothesis (c) $\mathcal{G} \models \mathcal{T}'.2 :: \mathcal{K}_2'[\alpha \leftarrow \mathcal{T}'.1]$. Hence, by definition of the logical relation, $\mathcal{G} \models \mathcal{T}' :: \Sigma(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ holds, and so $\mathcal{G} \models \mathcal{T}' :: \mathcal{K}'$ holds.

(d) Since $\mathcal{K}' \cap \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \neq \emptyset$, $\mathcal{K}' = \Sigma(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ where $\mathcal{K}_1' \cap \mathcal{K}_1 \neq \emptyset$ and $\mathcal{K}_2' \cap \mathcal{K}_2 \neq \emptyset$. As shown in (b) we have $\mathcal{G} \models \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \cup \Sigma(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$. Since $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ holds by definition, we get by induction hypothesis (d) $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1 \cup \mathcal{K}_1'$. And since $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$ holds by definition, we get by induction hypothesis (d) $\mathcal{G} \models \mathcal{T}.2 :: (\mathcal{K}_2 \cup \mathcal{K}_2')[\alpha \leftarrow \tau.1]$. Hence, by definition of the logical relation, $\Gamma \models \mathcal{T} :: \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2 \cup \Sigma(\alpha : \mathcal{K}_1')\,\mathcal{K}_2'$ holds, and so $\Gamma \models \mathcal{T} :: \mathcal{K} \cup \mathcal{K}'$ holds.

(e) We have $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ and $\mathcal{G} \models \mathcal{T}'.1 :: \mathcal{K}_1$, hence by induction hypothesis (e) we get $\mathcal{G} \models \mathcal{T}.1 \cup \mathcal{T}'.1 :: \mathcal{K}_1$. Then, by definition of the logical relation, $\mathcal{G} \models \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \mathcal{T}').1]$ holds. Then, because we have $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$, we get $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \mathcal{T}').1]$ by induction hypothesis (d). Similarly, $\mathcal{G} \models \mathcal{T}'.2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \mathcal{T}').1]$ holds. Then, by induction hypothesis (e) we get $\mathcal{G} \models (\mathcal{T} \cup \mathcal{T}').2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \mathcal{T}').1]$. Hence, by definition of the logical relation, we have $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. □

A direct consequence of the last lemma is that the logical relation is stable by taking super-sets.

Lemma 3.5.32. *If* $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ *and* $\mathcal{G} \models \mathcal{K}'$ *and* $\mathcal{K} \subseteq \mathcal{K}'$ *then* $\mathcal{G} \models \mathcal{T} :: \mathcal{K}'$.

Proof. Using Lemma 3.5.31 on page 87, item(d). □

The next lemmas shows that the relation involving a set of types $\mathcal{T}$ can be extended by a set of types $\mathcal{T}'$ such that $\mathcal{T}$ and $\mathcal{T}'$ are pointwise related with respect to convertibility up to insertion of expansors (that is, with respect to the relation used for the base case of the logical relation).

Lemma 3.5.33. *Assume $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ and for all $\Gamma \in \mathcal{G}$ and $\tau' \in \mathcal{T}'$, there exists $\tau \in \mathcal{T}$ such that $\lceil \tau' \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil \Gamma \rceil}$ (we say that $\mathcal{T}'$ is convertible with $\mathcal{T}$ in $\mathcal{G}$). Then $\mathcal{G} \models \mathcal{T} \cup \mathcal{T}' :: \mathcal{K}$.*

Proof. By induction on the size of sets of kinds:

- $\kappa = \{\star\}$: Let $\tau_1, \tau_2 \in \mathcal{T} \cup \mathcal{T}'$ and $\Gamma \in \mathcal{G}$. There are four cases:
  - $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$: then, since $\Gamma \models \mathcal{T} :: \{\star\}$, $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds by assumption.
  - $\tau_1 \in \mathcal{T}'$ and $\tau_2 \in \mathcal{T}$: there exists $\tau_3 \in \mathcal{T}$ such that $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$. Moreover, $\lceil \tau_2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$ holds by assumption. We conclude by symmetry and transitivity of the convertibility relation.
  - $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}'$: similar to the previous case.
  - $\tau_1 \in \mathcal{T}'$ and $\tau_2 \in \mathcal{T}'$: there exists $\tau_3, \tau_4 \in \mathcal{T}$ such that $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_3 \rceil^{\lceil \Gamma \rceil}$ and $\lceil \tau_2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_4 \rceil^{\lceil \Gamma \rceil}$. Moreover, $\lceil \tau_3 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_4 \rceil^{\lceil \Gamma \rceil}$ holds by assumption. We conclude by symmetry and transitivity.
- $\kappa = \mathcal{S}(\mathcal{U})$: we have $\mathcal{G} \models \mathcal{S}(\mathcal{U})$ and $\mathcal{G} \models \mathcal{T} :: \mathcal{S}(\mathcal{U})$ with $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$. Then $\mathcal{T}'$ is convertible with $\mathcal{T} \cup \mathcal{U}$ in $\mathcal{G}$. Hence, by induction hypothesis, $\mathcal{G} \models \mathcal{T}' \cup \mathcal{T} \cup \mathcal{U} :: \{\star\}$. Hence $\mathcal{G} \models \mathcal{T}' \cup \mathcal{T} :: \mathcal{S}(\mathcal{U})$.
- $\kappa = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$: we have $\mathcal{G} \models \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and $\mathcal{T}_1$ such that $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$. Then $\mathcal{G}' \models \mathcal{T}\ \mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$ holds by definition. Then, $\mathcal{T}'\ \mathcal{T}_1$ is convertible with $\mathcal{T}\ \mathcal{T}_1$ in $\mathcal{G}$, hence also in $\mathcal{G}'$. By induction hypothesis, $\mathcal{G}' \models (\mathcal{T}'\ \mathcal{T}_1) \cup (\mathcal{T}\ \mathcal{T}_1) :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$. Finally, since $(\mathcal{T}'\ \mathcal{T}_1) \cup (\mathcal{T}\ \mathcal{T}_1) = (\mathcal{T}' \cup \mathcal{T})\ \mathcal{T}_1$, we get $\mathcal{G} \models \mathcal{T}' \cup \mathcal{T} :: \Pi(\mathcal{K}_1 : \alpha)\,\mathcal{K}_2$ by definition of the logical relation.
- $\kappa = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$: we have $\mathcal{G} \models \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ and $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$. Then, since $\mathcal{T}'.1$ is convertible with $\mathcal{T}.1$ in $\mathcal{G}$, we have by induction hypothesis $\mathcal{G} \models (\mathcal{T}' \cup \mathcal{T}).1 :: \mathcal{K}_1$. Hence, by definition of the logical relation, $\mathcal{G} \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}'.1 \cup \mathcal{T}.1]$ holds. Also, since $\mathcal{T}'.2$ is convertible with $\mathcal{T}.2$ in $\mathcal{G}$, we have by induction hypothesis $\mathcal{G} \models (\mathcal{T}' \cup \mathcal{T}).2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$. Since $\mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1] \subseteq \mathcal{K}_2[\alpha \leftarrow \mathcal{T}'.1 \cup \mathcal{T}.1]$, we conclude by Lemma 3.5.32 on the previous page that $\mathcal{G} \models (\mathcal{T}' \cup \mathcal{T}).2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T}' \cup \mathcal{T}).1]$. Hence $\mathcal{G} \models \mathcal{T} :: \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. □

The following lemma is specific to our proof: it is absent from [SH06]. It states that the logical relation is stable under head expansion.

Lemma 3.5.34. *If $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$, then $\mathcal{G} \models \mathcal{T} \cup \eta_{\mathcal{K}}\ \mathcal{T} :: \mathcal{K}$.*

Proof. By induction on the sizes of the sets of kinds:

- $\mathcal{K} = \{\star\}$: immediate, since $\lceil \eta_\star\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil \Gamma \rceil}$.
- $\mathcal{K} = \mathcal{S}(\mathcal{U})$: By definition, we get $\Gamma \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$, which means that: for all $\Gamma \mathrm{in} \mathcal{G}$ and $\tau_1, \tau_2 \mathrm{in} \mathcal{T} \cup \mathcal{U}$, $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$ holds. Let us call (H) this result. We want to prove that $\mathcal{G} \models \mathcal{T} \cup \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T} :: \mathcal{S}(\mathcal{U})$ holds. By definition, it suffices to prove that $\mathcal{G} \models \mathcal{T} \cup \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T} \cup \mathcal{U} :: \{\star\}$ holds. Hence, by definition again, it suffices to prove that for every Γ in $\mathcal{G}$ and every $\tau_1, \tau_2 \in \mathcal{T} \cup \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T} \cup \mathcal{U}$, we have $\lceil \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$. There are four cases:
  1. $\tau_1 \in \mathcal{T} \cup \mathcal{U}$ and $\tau_2 \in \mathcal{T} \cup \mathcal{U}$: this is proved by (H).

2. $\tau_1 \in \mathcal{T} \cup \mathcal{U}$ and $\tau_2 \in \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T}$: then $\tau_2 = \eta_{\mathcal{S}(u_2')}\ \tau_2'$ for some $u_2' \in \mathcal{U}$ and $\tau_2' \in \mathcal{T}$. Since $\lceil \eta_{\mathcal{S}(u_2')}\ \tau_2' \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil u_2' \rceil^{\lceil \Gamma \rceil}$, we conclude by (H) and transitivity.
3. $\tau_1 \in \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T}$ and $\tau_2 \in \mathcal{T} \cup \mathcal{U}$: this is symmetric to the previous case.
4. $\tau_1 \in \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T}$ and $\tau_2 \in \eta_{\mathcal{S}(\mathcal{U})}\ \mathcal{T}$: then $\tau_1 = \eta_{\mathcal{S}(u_1')}\ \tau_1'$ for some $u_1' \in \mathcal{U}$ and $\tau_1' \in \mathcal{T}$ and $\tau_2 = \eta_{\mathcal{S}(u_2')}\ \tau_2'$ for some $u_2' \in \mathcal{U}$ and $\tau_2' \in \mathcal{T}$. Then $\lceil \mathcal{S}(u_1')\ \tau_1' \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil u_1' \rceil^{\lceil \Gamma \rceil}$ and $\lceil \mathcal{S}(u_2')\ \tau_2' \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil u_2' \rceil^{\lceil \Gamma \rceil}$ hold. We also get from (H) that $\lceil u_1' \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil u_2' \rceil^{\lceil \Gamma \rceil}$. We conclude by transitivity.

- $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$: let $\Gamma' \sqsupseteq \Gamma$ and $\mathcal{T}_1$ such that $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$. By induction hypothesis, $\mathcal{G}' \models \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1 :: \mathcal{K}_1$. Hence $\mathcal{G}' \models \mathcal{T}\ (\mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1) :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1]$. Hence by induction hypothesis, $\mathcal{G}' \models (\mathcal{T}\ (\mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1)) \cup (\eta_{\mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1]}\ \mathcal{T}\ (\mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1)) :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1]$. Hence by Lemma 3.5.31 on page 87(c), we get $\mathcal{G}' \models \mathcal{T}\ \mathcal{T}_1 \cup \eta_{\mathcal{K}_2[\alpha \leftarrow \eta_{\mathcal{K}_1}\ \mathcal{T}_1]}\ \mathcal{T}\ (\eta_{\mathcal{K}_1}\ \mathcal{T}_1) :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$. Then, since $\mathcal{T}\ \mathcal{T}_1 \cup (\eta_{\Pi(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T})\ \mathcal{T}_1$ is convertible with $\mathcal{T}\ \mathcal{T}_1 \cup \eta_{\mathcal{K}_2[\alpha \leftarrow \eta_{\mathcal{K}_1}\ \mathcal{T}_1]}\ \mathcal{T}\ (\eta_{\mathcal{K}_1}\ \mathcal{T}_1)$ in $\mathcal{G}'$, by applying Lemma 3.5.33 on the preceding page, we get $\mathcal{G}' \models \mathcal{T}\ \mathcal{T}_1 \cup (\eta_{\Pi(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T})\ \mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$, hence $\mathcal{G}' \models (\mathcal{T} \cup \eta_{\Pi(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T})\ \mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$. Hence, by the definition of the logical relation, $\mathcal{G}' \models \mathcal{T} \cup \eta_{\Pi(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T} :: \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$.
- $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$: by induction hypothesis, $\mathcal{G} \models \mathcal{T}.1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}.1 :: \mathcal{K}_1$ holds. Since $\mathcal{T}.1 \cup (\eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \tau).1$ is convertible with $\mathcal{T}.1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}.1$ in $\mathcal{G}$, by Lemma 3.5.33 on the facing page we get $\mathcal{G} \models (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1 :: \mathcal{K}_1$. Hence, by definition of the logical relation, $\mathcal{G} \models \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]$. Then, by Lemma 3.5.32 on page 89, we get $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]$. Then, by induction hypothesis, we have $\mathcal{G} \models \mathcal{T}.2 \cup \eta_{\mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]}\ \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]$. Then by Lemma 3.5.31 on page 87(c), $\mathcal{G} \models \mathcal{T}.2 \cup \eta_{\mathcal{K}_2[\alpha \leftarrow (\eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]}\ \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]$. And since $\mathcal{T}.2 \cup (\eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).2$ is convertible with $\mathcal{T}.2 \cup \eta_{\mathcal{K}_2[\alpha \leftarrow (\eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]}\ \mathcal{T}.2$ in $\mathcal{G}$, we have by Lemma 3.5.33 on the preceding page $\mathcal{G} \models (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).2 :: \mathcal{K}_2[\alpha \leftarrow (\mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T}).1]$. Hence, by definition of the logical relation, $\mathcal{G} \models \mathcal{T} \cup \eta_{\Sigma(\alpha:\mathcal{K}_1)\,\mathcal{K}_2}\ \mathcal{T} :: \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. □

We can eventually show that the logical relation entails our desired property on convertibility, that is: types (*resp.* kinds) in the relation are convertible up to insertion of expansors, preceded by head expansion.

Proposition 3.5.35. *The following assertions hold:*

(a) If $\mathcal{G} \models \mathcal{K}$, then for all $\Gamma \in \mathcal{G}$ and $\kappa_1, \kappa_2 \in \mathcal{K}$, $\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_2 \rceil^{\lceil \Gamma \rceil}$;

(b) If $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$, then for all $\Gamma \in \mathcal{G}$ and $\kappa \in \mathcal{K}$ and $\tau_1, \tau_2 \in \mathcal{T}$, $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil \Gamma \rceil}$.

(c) If $\mathcal{G} \models \mathcal{K}$ and if there is a set $\mathcal{T}$ such that for all $\Gamma \in \mathcal{G}$ and $\tau_1, \tau_2 \in \mathcal{T}$, there exists $\kappa \in \mathcal{K}$ such that $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}$, then $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$.

Proof. By induction on the size of sets of kinds:

- $\mathcal{K} = \{\star\}$:
  (a) $\lceil \star \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \star \rceil^{\lceil \Gamma \rceil}$ holds by reflexivity.
  (b) Immediate, since $\lceil \eta_\star\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil \Gamma \rceil}$.
  (c) Immediate, since $\lceil \eta_\star\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil \Gamma \rceil}$.
- $\mathcal{K} = \mathcal{S}(\mathcal{U})$:

(a) By induction hypothesis (b), for all $\Gamma \in \mathcal{G}$ and $u_1, u_2 \in \mathcal{U}$, $\lceil \eta_\star\ u_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\star\ u_2 \rceil^{\lceil\Gamma\rceil}$, hence $\lceil \mathcal{S}(u_1) \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \mathcal{S}(u_2) \rceil^{\lceil\Gamma\rceil}$.

(b) $\lceil \eta_{\mathcal{S}(u)}\ \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\mathcal{S}(u)}\ \tau_2 \rceil^{\lceil\Gamma\rceil}$ holds by reflexivity.

(c) Let $\tau_1, \tau_2 \in \mathcal{T} \cup \mathcal{U}$. There are four cases:

- $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$: By assumption, we have for all $\Gamma \in \mathcal{G}$ and $\tau_1', \tau_1' \in \mathcal{T}$, there exists $\mathcal{S}(u') \in \mathcal{S}(\mathcal{U})$ such that $\lceil \eta_{\mathcal{S}(\tau_1')}\ \Gamma \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2' \rceil^{\lceil\Gamma\rceil}$. As a consequence, by instantiating a first on τ_1 and τ_1, and a second time on τ_2 and τ_2, we know that there exists $u_1, u_2 \in \mathcal{U}$ such that $\lceil \eta_{\mathcal{S}(u_1)}\ \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_1 \rceil^{\lceil\Gamma\rceil}$ and $\lceil \eta_{\mathcal{S}(u_2)}\ \tau_2 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil\Gamma\rceil}$. And by induction hypothesis (b), $\lceil \eta_\star\ u_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\star\ u_2 \rceil^{\lceil\Gamma\rceil}$, hence $\lceil \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil\Gamma\rceil}$.
- $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{U}$: there exists $u_1 \in \mathcal{U}$ such that $\lceil \eta_{\mathcal{S}(u_1)}\ \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_1 \rceil^{\lceil\Gamma\rceil}$. And by induction hypothesis (b), $\lceil \eta_\star\ u_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\star\ \tau_2 \rceil^{\lceil\Gamma\rceil}$, hence $\lceil \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil\Gamma\rceil}$.
- $\tau_1 \in \mathcal{U}$ and $\tau_2 \in \mathcal{T}$: similar to the previous case.
- $\tau_1 \in \mathcal{U}$ and $\tau_2 \in \mathcal{U}$: by induction hypothesis (b), $\lceil \eta_\star\ \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\star\ \tau_2 \rceil^{\lceil\Gamma\rceil}$.

Hence $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \star$, hence $\mathcal{G} \models \mathcal{T} :: \mathcal{S}(\mathcal{U})$.

- $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$:

(a) Let $\Pi(\alpha : \kappa_1)\, \kappa_2, \Pi(\alpha : \kappa_1')\, \kappa_2' \in \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$ and $\Gamma \in \mathcal{G}$. By induction hypothesis (a), $\lceil \kappa_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_1' \rceil^{\lceil\Gamma\rceil}$. Then let $\mathcal{G}' = \mathcal{G}, \alpha :: \mathcal{K}_1$. $\mathcal{G}' \sqsupseteq \mathcal{G}$ holds. Then, for all $\Gamma' \in \mathcal{G}$, $\lceil \eta_{\Gamma'(\alpha)}\ \alpha \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \alpha \rceil^{\lceil\Gamma'\rceil}$ holds by Lemma 3.5.19 on page 82. Hence $\mathcal{G}' \models \{\alpha\} :: \mathcal{K}_1$ by induction hypothesis (c). Then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \{\alpha\}]$ by definition of the logical relation. Then $\lceil \kappa_2 \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_2' \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil}$ by induction hypothesis (a). Finally, $\lceil \Pi(\alpha : \kappa_1)\, \kappa_2 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \Pi(\alpha : \kappa_1')\, \kappa_2' \rceil^{\lceil\Gamma\rceil}$.

(b) Let $\mathcal{G}' = \mathcal{G}, \alpha :: \mathcal{K}_1$. $\mathcal{G}' \sqsupseteq \mathcal{G}$ holds. Then, for all $\Gamma' \in \mathcal{G}$, $\lceil \eta_{\Gamma'(\alpha)}\ \alpha \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \alpha \rceil^{\lceil\Gamma'\rceil}$ holds by Lemma 3.5.19 on page 82. Hence $\mathcal{G}' \models \{\alpha\} :: \mathcal{K}_1$ by induction hypothesis (c). Then $\mathcal{G}' \models \mathcal{T}\ \{\alpha\} :: \mathcal{K}_2[\alpha \leftarrow \{\alpha\}]$ by definition of the logical relation. Let $\tau, \tau' \in \mathcal{T}$ and $\Pi(\alpha : \kappa_1)\, \kappa_2 \in \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$ and $\Gamma \in \mathcal{G}$. By induction hypothesis (b), we have $\lceil \eta_{\kappa_2}\ (\tau\ \alpha) \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_2}\ (\tau'\ \alpha) \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil}$. Hence $\lceil \eta_{\Pi(\alpha:\kappa_1)\, \kappa_2}\ \tau \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lambda(\alpha :: \lceil \kappa_1 \rceil^{\lceil\Gamma\rceil})\, \lceil \eta_{\kappa_2}\ (\tau\ \alpha) \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lambda(\alpha :: \lceil \kappa_1 \rceil^{\lceil\Gamma\rceil})\, \lceil \eta_{\kappa_2}\ (\tau'\ \alpha) \rceil^{\lceil\Gamma, \alpha :: \kappa_1\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\Pi(\alpha:\kappa_1)\, \kappa_2}\ \tau' \rceil^{\lceil\Gamma\rceil}$.

(c) Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and $\mathcal{T}_1$ such that $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$. Let $\Gamma' \in \mathcal{G}'$ and $(\tau\ \tau_1), (\tau'\ \tau_1') \in \mathcal{T}\ \mathcal{T}_1$. By assumption, there exists $\Pi(\alpha : \kappa_1)\, \kappa_2 \in \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$ such that $\lceil \eta_{\Pi(\alpha:\kappa_1)\, \kappa_2}\ \tau \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil\Gamma'\rceil}$ holds. Similarly, there exists a kind $\Pi(\alpha : \kappa_1')\, \kappa_2' \in \Pi(\alpha : \mathcal{K}_1)\, \mathcal{K}_2$ such that $\lceil \eta_{\Pi(\alpha:\kappa_1')\, \kappa_2'}\ \tau \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau' \rceil^{\lceil\Gamma'\rceil}$ holds. By induction hypothesis (a) we get $\lceil \kappa_1 \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_1' \rceil^{\lceil\Gamma'\rceil}$. Moreover $\lceil \eta_{\kappa_1}\ \tau_1 \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_1}\ \tau_1' \rceil^{\lceil\Gamma'\rceil}$ by induction hypothesis (b). Moreover, $\mathcal{G}' \models \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1 :: \mathcal{K}_1$ holds by Lemma 3.5.34 on page 90. Hence by Lemma 3.5.30 on page 87 and the definition of the logical relation, $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}_1]$ holds, hence we get $\lceil \kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1] \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_2'[\alpha \leftarrow \eta_{\kappa_1'}\ \tau_1'] \rceil^{\lceil\Gamma'\rceil}$ and $\lceil \kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1] \rceil^{\lceil\Gamma'\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow}$

$\lceil \kappa_2[\alpha \leftarrow \tau_1] \rceil^{\lceil \Gamma' \rceil}$ by induction hypothesis (a). Then:

$$\begin{array}{rcl}
\lceil \tau'\ \tau_1' \rceil^{\lceil \Gamma' \rceil} & \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\Pi(\alpha:\kappa_1')\,\kappa_2'}\ \tau\ \tau_1' \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2'[\alpha \leftarrow \eta_{\kappa_1'}\ \tau_1']}\ (\tau\ (\eta_{\kappa_1'}\ \tau_1')) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\tau\ (\eta_{\kappa_1'}\ \tau_1')) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\tau\ (\eta_{\kappa_1}\ \tau_1)) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\tau\ (\eta_{\kappa_1}\ \tau_1))) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ \tau\ \tau_1) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau_1]}\ (\tau\ \tau_1) \rceil^{\lceil \Gamma' \rceil} \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\kappa_2[\alpha \leftarrow \tau_1]}\ (\tau\ \tau_1) \rceil^{\lceil \Gamma' \rceil}
\end{array}$$

We just proved that for all $\Gamma' \in \mathcal{G}'$ and for all $\tau, \tau' \in \mathcal{T}\ \mathcal{T}_1$, there exists $\kappa \in \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$ such that $\lceil \eta_\kappa\ \tau \rceil^{\lceil \Gamma' \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \tau' \rceil^{\lceil \Gamma' \rceil}$. Then by induction hypothesis (c), we get $\mathcal{G}' \models \mathcal{T}\ \mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$. Hence $\mathcal{G} \models \mathcal{T} :: \Pi(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ by definition of the logical relation.

- $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$:

  (a) Let $\Sigma(\alpha : \kappa_1)\,\kappa_2, \Sigma(\alpha : \kappa_1')\,\kappa_2' \in \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ and $\Gamma \in \mathcal{G}$. By induction hypothesis (a), $\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \kappa_1' \rceil^{\lceil \Gamma \rceil}$. Then let $\mathcal{G}' = \mathcal{G}, \alpha :: \mathcal{K}_1$. $\mathcal{G}' \sqsupseteq \mathcal{G}$ holds. Then, for all $\Gamma' \in \mathcal{G}$, $\lceil \eta_{\Gamma'(\alpha)}\ \alpha \rceil^{\lceil \Gamma' \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \alpha \rceil^{\lceil \Gamma' \rceil}$ holds by Lemma 3.5.19 on page 82. Hence $\mathcal{G}' \models \{\alpha\} :: \mathcal{K}_1$ by induction hypothesis (c). Then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \{\alpha\}]$ by definition of the logical relation. Then $\lceil \kappa_2 \rceil^{\lceil \Gamma, \alpha :: \kappa_1 \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \kappa_2' \rceil^{\lceil \Gamma, \alpha :: \kappa_1 \rceil}$ by induction hypothesis (a). Finally, $\lceil \Sigma(\alpha : \kappa_1)\,\kappa_2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \Sigma(\alpha : \kappa_1')\,\kappa_2' \rceil^{\lceil \Gamma \rceil}$.

  (b) Let $\Gamma \in \mathcal{G}, \tau, M' \in \mathcal{T}$ and $\Sigma(\alpha : \kappa_1)\,\kappa_2 \in \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. By induction hypothesis (b) we get $\lceil \eta_{\kappa_1}\ \tau.1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \eta_{\kappa_1}\ \tau'.1 \rceil^{\lceil \Gamma \rceil}$. Moreover, $\mathcal{G} \models \mathcal{T}.1 \cup \mathcal{K}_1\ \mathcal{T}.1 :: \mathcal{K}_1$ by Lemma 3.5.34 on page 90, then $\mathcal{G} \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1 \cup \mathcal{K}_1\ \mathcal{T}.1]$ by definition of the logical relation, and then $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1 \cup \mathcal{K}_1\ \mathcal{T}.1]$ by Lemma 3.5.32 on page 89. Hence by induction hypothesis (b) we get $\lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau'.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau'.1]}\ \tau'.2 \rceil^{\lceil \Gamma \rceil}$. Induction hypothesis (a) also gives $\lceil \kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau.1] \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau'.1] \rceil^{\lceil \Gamma \rceil}$. Then:

$$\begin{array}{rcl}
\lceil \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau \rceil^{\lceil \Gamma \rceil} & \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & (\lceil \eta_{\kappa_1}\ \tau.1 \rceil^{\lceil \Gamma \rceil}, \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil}) \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & (\lceil \eta_{\kappa_1}\ \tau'.1 \rceil^{\lceil \Gamma \rceil}, \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil}) \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & (\lceil \eta_{\kappa_1}\ \tau'.1 \rceil^{\lceil \Gamma \rceil}, \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau'.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil}) \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & (\lceil \eta_{\kappa_1}\ \tau'.1 \rceil^{\lceil \Gamma \rceil}, \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau'.1]}\ \tau'.2 \rceil^{\lceil \Gamma \rceil}) \\
& \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet & \lceil \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau' \rceil^{\lceil \Gamma \rceil}
\end{array}$$

  (c) Let $\Gamma \in \mathcal{G}$ and $\tau.1, \tau'.1 \in \mathcal{T}.1$. There exists $\Sigma(\alpha : \kappa_1)\,\kappa_2 \in \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ such that $\lceil \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \tau' \rceil^{\lceil \Gamma \rceil}$ holds. Then $\lceil \tau'.1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil (\eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau).1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \eta_{\kappa_1}\ \tau.1 \rceil^{\lceil \Gamma \rceil}$ holds. Hence, by induction hypothesis (c), $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ holds. As a consequence of Lemma 3.5.34 on page 90, $\mathcal{G} \models \mathcal{T}.1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}.1 :: \mathcal{K}_1$ also holds, hence by definition of the logical relation, $\mathcal{G} \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1 \cup \eta_{\mathcal{K}_1}\ \mathcal{T}.1]$ holds. Let $\Gamma \in \mathcal{G}$ and $\tau.2, \tau'.2 \in \mathcal{T}.2$. There exists $\Sigma(\alpha : \kappa_1)\,\kappa_2 \in \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$ such that $\lceil \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \tau' \rceil^{\lceil \Gamma \rceil}$. Then by induction hypothesis (a), we get $\lceil \kappa_2[\alpha \leftarrow \tau.1] \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta}{\longleftrightarrow}_\bullet \lceil \kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau.1] \rceil^{\lceil \Gamma \rceil}$ Then

$\lceil \tau'.2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil (\eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau).2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1}\ \tau.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_2[\alpha \leftarrow \tau.1]}\ \tau.2 \rceil^{\lceil \Gamma \rceil}$. Hence by induction hypothesis (c) we get $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$. Finally, by definition of the logical relation, $\mathcal{G} \models \mathcal{T} :: \Sigma(\alpha : \mathcal{K}_1)\,\mathcal{K}_2$. □

The next lemma permits to consider a valid extension of a set of substitution, under suitable conditions. This will be necessary to prove Lemma 3.5.37 in the case of functions.

Lemma 3.5.36 (Valid substitution extension)**.** *If* $\mathcal{G} \models \mathfrak{S} : \Gamma$*, and* $\Gamma, \alpha :: \kappa \vdash_{\mathsf{HS}} \text{ok}$ *and* $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\kappa)$ *and* $\alpha \notin \operatorname{dom} \mathfrak{S}$*, then* $\mathcal{G} \models \mathfrak{S}; \alpha \mapsto \mathcal{T} : \Gamma, \alpha :: \kappa$.

Proof. We have $\alpha \notin \operatorname{dom}\Gamma \cup \mathrm{fv}(\Gamma) \cup \mathrm{fv}(\kappa)$. Let $\beta \in \operatorname{dom}\Gamma, \alpha :: \kappa$. There are two cases:

- $\underline{\alpha \neq \beta}$: then $\beta \in \operatorname{dom}\Gamma$, $(\mathfrak{S}, \alpha \mapsto \mathcal{T})(\beta) = \mathfrak{S}(\beta)$ and $(\mathfrak{S}; \alpha \mapsto \mathcal{T})((\Gamma, \alpha :: \kappa)(\beta)) = (\mathfrak{S}; \alpha \mapsto \kappa)(\Gamma(\beta)) = \mathfrak{S}(\Gamma(\beta))$. Hence $\mathcal{G} \models \mathfrak{S}(\beta) :: \mathfrak{S}(\Gamma(\beta))$ holds by assumption.
- $\underline{\alpha = \beta}$: then $(\mathfrak{S}, \alpha \mapsto \mathcal{T})(\alpha) = \mathcal{T}$ and $(\mathfrak{S}; \alpha \mapsto \mathcal{T})((\Gamma, \alpha :: \kappa)(\alpha)) = (\mathfrak{S}; \alpha \mapsto \mathcal{T})(\kappa) = \mathfrak{S}(\kappa)$. Hence $\mathcal{G} \models \mathfrak{S}(\alpha) :: \mathfrak{S}(\Gamma(\alpha))$ holds by assumption.

Then for all $\beta \in \operatorname{dom}\Gamma, \alpha :: \kappa$, we have $\mathcal{G} \models \mathfrak{S}(\beta) :: \mathfrak{S}(\Gamma(\beta))$. □

Now comes the fundamental completeness lemma, that relates the wellformedness judgments with the logical relation.

Lemma 3.5.37 (Fundamental completeness lemma)**.** *The following assertions hold:*

1. *if* $\Gamma \vdash_{\mathsf{HS}} \kappa$ ok *then* $\mathcal{G} \models \mathfrak{S} : \Gamma$ *implies* $\mathcal{G} \models \mathfrak{S}(\kappa)$*;*
2. *if* $\Gamma \vdash_{\mathsf{HS}} \kappa \equiv \kappa'$ *then* $\mathcal{G} \models \mathfrak{S} : \Gamma$ *implies* $\mathcal{G} \models \mathfrak{S}(\kappa) \cup \mathfrak{S}(\kappa')$*;*
3. *if* $\Gamma \vdash_{\mathsf{HS}} \kappa \leqslant \kappa'$ *then* $\mathcal{G} \models \mathfrak{S} : \Gamma$ *implies:*
   - $\mathcal{G} \models \mathfrak{S}(\kappa)$*;*
   - $\mathcal{G} \models \mathfrak{S}(\kappa')$*;*
   - *if* $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\kappa)$*, then* $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\kappa')$*;*
4. *if* $\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa$ *then* $\mathcal{G} \models \mathfrak{S} : \Gamma$ *implies* $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\kappa)$*;*
5. *if* $\Gamma \vdash_{\mathsf{HS}} \tau \equiv \tau' :: \kappa$ *then* $\mathcal{G} \models \mathfrak{S} : \Gamma$ *implies* $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') :: \mathfrak{S}(\kappa)$*;*

Proof. By induction on the height of the judgments (the proof is the same as Stone-Harper's, except for cases Lam and EqLam, which use Lemma 3.5.34 on page 90):

Kind wellformedness judgment:

- $\underline{\text{WfKindStar:}}$ $\kappa = \star$. Hence $\mathfrak{S}(\kappa) = \{\star\}$ and $\mathcal{G} \models \{\star\}$ holds by definition of the logical relation.
- $\underline{\text{WfKindSingle:}}$ $\kappa = \mathcal{S}(\tau)$. By induction hypothesis (4), $\mathcal{G} \models \mathfrak{S}(\tau) :: \{\star\}$ holds. Hence $\mathcal{G} \models \mathcal{S}(\mathfrak{S}(\tau))$ by definition of the logical relation. Since $\mathcal{S}(\mathfrak{S}(\tau)) = \mathfrak{S}(\mathcal{S}(\tau))$, $\mathcal{G} \models \mathfrak{S}(\mathcal{S}(\tau))$ holds.
- $\underline{\text{WfKindPi:}}$ $\kappa = \Pi(\alpha : \kappa_1)\,\kappa_2$. We have $\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2$ ok, therefore there is a strict sub-derivation $\Gamma \vdash_{\mathsf{HS}} \kappa_1$ ok. Hence, by induction hypothesis (1) we have $\mathcal{G} \models \mathfrak{S}(\kappa_1)$. Now let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. By Lemma 3.5.30 on page 87 and Lemma 3.5.36, $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$ holds. Then, by induction hypothesis (1), we get $\mathcal{G}' \models \mathfrak{S}'(\kappa_2)$. Since $\alpha \notin \mathrm{fv}(\mathfrak{S})$, $\mathfrak{S}'(\kappa_2) = \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$. Hence $\mathcal{G}' \models \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$. Therefore, by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\Pi(\alpha : \kappa_1)\,\kappa_2)$ holds.
- $\underline{\text{WfKindSigma:}}$ similar to the previous case.

Kind equivalence judgment:

- EqStar: $\kappa = \kappa' = \star$. Since $\mathcal{G} \models \{\star\}$ holds by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\kappa_1) \cup \mathfrak{S}(\kappa_2)$ holds.
- EqSingle: $\kappa = \mathcal{S}(\tau)$ and $\kappa' = \mathcal{S}(\tau')$. By induction hypothesis (5), $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') :: \{\star\}$ holds. Hence $\mathcal{G} \models \mathfrak{S}(\mathcal{S}(\tau)) \cup \mathfrak{S}(\mathcal{S}(\tau'))$ holds.
- EqPi: $\kappa = \Pi(\alpha : \kappa_1)\, \kappa_2$ and $\kappa' = \Pi(\alpha : \kappa_1')\, \kappa_2'$. By induction hypothesis (2), $\mathcal{G} \models \mathfrak{S}(\kappa_1) \cup \mathfrak{S}(\kappa_1')$ holds. Now let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1) \cup \mathfrak{S}(\kappa_1')$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. By Lemma 3.5.30 on page 87 and Lemma 3.5.36 on the facing page, $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$ holds. Then, by induction hypothesis (2), we get $\mathcal{G}' \models \mathfrak{S}'(\kappa_2) \cup \mathfrak{S}'(\kappa_2')$. Since $\alpha \notin \mathrm{fv}(\mathfrak{S})$, $\mathfrak{S}'(\kappa_2) = \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$ and $\mathfrak{S}'(\kappa_2') = \mathfrak{S}(\kappa_2')[\alpha \leftarrow \mathcal{T}_1]$. Hence $\mathcal{G}' \models (\mathfrak{S}(\kappa_2) \cup \mathfrak{S}'(\kappa_2'))[\alpha \leftarrow \mathcal{T}_1]$. Therefore, by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\Pi(\alpha : \kappa_1)\, \kappa_2) \cup \mathfrak{S}(\Pi(\alpha : \kappa_1')\, \kappa_2')$ holds.
- EqSigma: similar to the previous case.

Subkinding judgment:

- SubForget: $\kappa = \mathcal{S}(\tau)$ and $\kappa' = \star$.
  - $\mathcal{G} \models \mathfrak{S}(\star)$ always hold by definition.
  - Same proof as above.
  - Same proof as above.
- SubStar: $\kappa = \kappa' = \star$.
  - $\mathcal{G} \models \mathfrak{S}(\star)$ always hold by definition.
  - Same proof as above.
  - Same proof as above.
- SubSingle: $\kappa = \mathcal{S}(\tau)$ and $\kappa' = \mathcal{S}(\tau')$.
  - By induction hypothesis (5), $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') :: \{\star\}$ holds. Hence by Lemma 3.5.31 on page 87(c), $\mathcal{G} \models \mathfrak{S}(\tau) :: \{\star\}$ holds. Hence $\mathcal{G} \models \mathfrak{S}(\mathcal{S}(\tau))$ holds.
  - Same proof as above.
  - Same proof as above.
- SubPi: $\kappa = \Pi(\alpha : \kappa_1)\, \kappa_2$ and $\kappa' = \Pi(\alpha : \kappa_1')\, \kappa_2'$ with $\Gamma \vdash_{\mathsf{HS}} \kappa_1' \leqslant \kappa_1$ and $\Gamma, \alpha :: \kappa_1' \vdash_{\mathsf{HS}} \kappa_2 \leqslant \kappa_2'$ and $\Gamma \vdash_{\mathsf{HS}} \Pi(\alpha : \kappa_1)\, \kappa_2$ ok.
  - By induction hypothesis (1) we get $\mathcal{G} \models \mathfrak{S}(\Pi(\alpha : \kappa_1)\, \kappa_2)$.
  - By induction hypothesis (3), we get $\mathcal{G} \models \mathfrak{S}(\kappa_1')$. Let $\Gamma' \sqsupseteq \Gamma$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1')$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. By Lemma 3.5.30 on page 87 and Lemma 3.5.36 on the preceding page, we have $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1'$. Then by induction hypothesis (3) we get $\mathcal{G}' \models \mathfrak{S}'(\kappa_2')$, hence $\mathcal{G}' \models \mathfrak{S}(\kappa_2')[\alpha \leftarrow \mathcal{T}_1]$ since $\alpha \notin \mathrm{fv}(\mathfrak{S})$. Hence $\mathcal{G} \models \mathfrak{S}(\Pi(\alpha : \kappa_1')\, \kappa_2')$.
  - Assume $\mathcal{G} \models \tau :: \mathfrak{S}(\Pi(\alpha : \kappa_1)\, \kappa_2)$. Let $\Gamma' \sqsupseteq \Gamma$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1')$. Then by induction hypothesis (3), $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Then, by definition of the logical relation, $\mathcal{G}' \models \mathcal{T}\, \mathcal{T}_1 :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. We have $\mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1] = \mathfrak{S}'(\kappa_2)$ and $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1'$ by Lemma 3.5.30 on page 87 and Lemma 3.5.36 on the preceding page. Therefore, by induction hypothesis (3), we have $\mathcal{G}' \models \mathcal{T}\, \mathcal{T}_1 :: \mathfrak{S}(\kappa_2')[\alpha \leftarrow \mathcal{T}_1]$. As a consequence, $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\Pi(\alpha : \kappa_1)\, \kappa_2)$ holds.

- SubSigma: $\kappa = \Sigma(\alpha : \kappa_1)\,\kappa_2$ and $\kappa' = \Sigma(\alpha : \kappa_1')\,\kappa_2'$ with $\Gamma \vdash_{\mathsf{HS}} \kappa_1 \leqslant \kappa_1'$ and $\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \kappa_2 \leqslant \kappa_2'$ and $\Gamma \vdash_{\mathsf{HS}} \Sigma(\alpha : \kappa_1')\,\kappa_2'$ ok.
  - By induction hypothesis (3), we get $\mathcal{G} \models \mathfrak{S}(\kappa_1)$. Let $\Gamma' \sqsupseteq \Gamma$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. By lemmas Lemma 3.5.30 on page 87 and Lemma 3.5.36 on page 94 we have $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$. Then by induction hypothesis (3) we get $\mathcal{G}' \models \mathfrak{S}'(\kappa_2)$, hence $\mathcal{G}' \models \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$ since $\alpha \notin \mathrm{fv}(\mathfrak{S})$. Hence $\mathcal{G} \models \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$.
  - By induction hypothesis (1) we get $\mathcal{G} \models \mathfrak{S}(\Sigma(\alpha : \kappa_1')\,\kappa_2')$.
  - Assume $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$. By definition of the logical relation, we have $\mathcal{G} \models \mathcal{T}.1 :: \mathfrak{S}(\kappa_1)$. Then, by inductive hypothesis (3), $\mathcal{G} \models \mathcal{T}.1 :: \mathfrak{S}(\kappa_1')$ holds. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}.1$ we have $\mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}.1] = \mathfrak{S}'(\kappa_2)$ and $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$ by Lemma 3.5.30 on page 87 and Lemma 3.5.36 on page 94. Hence $\mathcal{G} \models \mathcal{T}.2 :: \mathfrak{S}'(\kappa_2)$ by definition of the logical relation. Then by induction hypothesis (3) we get $\mathcal{G} \models \mathcal{T}.2 :: \mathfrak{S}'(\kappa_2')$, and so $\mathcal{G} \models \mathcal{T}.2 :: \mathfrak{S}(\kappa_2')[\alpha \leftarrow \mathcal{T}.1]$ holds. Therefore we have $\mathcal{G} \models \mathcal{T} :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$.

Type wellformedness judgment:

- Var: $\tau = \alpha$ and $\kappa = \Gamma(\alpha)$. $\mathcal{G} \models \mathfrak{S}(\alpha) :: \mathfrak{S}(\kappa)$ holds by assumption.
- Lam: $\tau = \lambda(\alpha :: \kappa_1)\,\tau'$ and $\kappa = \Pi(\alpha : \kappa_1)\,\kappa_2$. There is a strict sub-derivation $\Gamma \vdash_{\mathsf{HS}} \kappa_1$ ok, hence by induction hypothesis (1), $\mathcal{G} \models \mathfrak{S}(\kappa_1)$. Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Then by Lemma 3.5.34 on page 90 we get $\mathcal{G}' \models \mathcal{T}_1 \cup \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1 \cup \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1$. By Lemma 3.5.30 on page 87 and Lemma 3.5.36 on page 94 we have $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$. Then, by induction hypothesis (4), we get $\mathcal{G}' \models \mathfrak{S}'(\tau') :: \mathfrak{S}'(\kappa_2)$, that is $\mathcal{G}' \models \mathfrak{S}(\tau')[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1] :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1 \cup \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1]$ since $\alpha \notin \mathrm{fv}(\mathfrak{S})$. Then by Lemma 3.5.31 on page 87(c), we have $\mathcal{G}' \models \mathfrak{S}(\tau')[\alpha \leftarrow \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1] :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$. Then $(\lambda(\alpha :: \mathfrak{S}(\kappa_1))\,\mathfrak{S}(\tau'))\,\mathcal{T}_1$ is convertible with $\mathfrak{S}(\tau')[\alpha \leftarrow \eta_{\mathfrak{S}(\kappa_1)}\,\mathcal{T}_1]$ in $\mathcal{G}'$, hence by Lemma 3.5.33 on page 90 and Lemma 3.5.31 on page 87(c), we have $\mathcal{G}' \models \mathfrak{S}(\lambda(\alpha :: \kappa_1)\,\tau')\,\mathcal{T}_1 :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$. Hence $\mathcal{G}' \models \mathfrak{S}(\lambda(\alpha :: \kappa_1)\,\tau') :: \mathfrak{S}(\Pi(\alpha : \kappa_1)\,\kappa_2)$ holds.
- App: $\tau = \tau_1\,\tau_2$ and $\kappa = \Pi(\alpha : \kappa_2)\,\kappa_1$. By induction hypothesis (4) we have $\mathcal{G} \models \mathfrak{S}(\tau_1) :: \mathfrak{S}(\Pi(\alpha : \kappa_2)\,\kappa_1)$ and $\mathcal{G} \models \mathfrak{S}(\tau_2) :: \mathfrak{S}(\kappa_2)$. Then, by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\tau_1)\,\mathfrak{S}(\tau_2) :: \mathfrak{S}(\kappa_1)[\alpha \leftarrow \mathfrak{S}(\tau_2)]$ holds, hence $\mathcal{G} \models \mathfrak{S}(\tau_1\,\tau_2) :: \mathfrak{S}(\kappa_1[\alpha \leftarrow \tau_2])$ by Lemma 3.5.31 on page 87(c).
- Pair: $\tau = (\tau_1, \tau_2)$ and $\kappa = \Sigma(\alpha : \kappa_1)\,\kappa_2$. By induction hypothesis (4), $\mathcal{G} \models \mathfrak{S}(\tau_1) :: \mathfrak{S}(\kappa_1)$ and $\mathcal{G} \models \mathfrak{S}(\tau_2) :: \mathfrak{S}(\kappa_2[\alpha \leftarrow \tau_1])$. And by induction hypothesis (1), $\mathcal{G} \models \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$. Then, because $(\mathfrak{S}(\tau_1, \tau_2)).1$ is convertible with $\mathfrak{S}(\mathcal{T}_1)$ in $\mathcal{G}$, we get by Lemma 3.5.33 on page 90 $\mathcal{G} \models (\mathfrak{S}(\tau_1, \tau_2)).1 \cup \mathfrak{S}(\tau_1) :: \mathfrak{S}(\kappa_1)$ holds. Similarly, $\mathcal{G} \models (\mathfrak{S}(\tau_1, \tau_2)).2 \cup \mathfrak{S}(\tau_2) :: \mathfrak{S}(\kappa_2[\alpha \leftarrow \tau_1])$ holds. Then, by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\kappa_2)[\alpha \leftarrow (\mathfrak{S}(\tau_1, \tau_2)).1 \cup \mathfrak{S}(\tau_1)]$ holds. Hence, by Lemma 3.5.32 on page 89, $\mathcal{G} \models (\mathfrak{S}(\tau_1, \tau_2)).2 \cup \mathfrak{S}(\tau_2) :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow (\mathfrak{S}(\tau_1, \tau_2)).1 \cup \mathfrak{S}(\tau_1)]$ holds. Then by applying Lemma 3.5.31 on page 87(c), we have $\mathcal{G} \models (\mathfrak{S}(\tau_1, \tau_2)).1 :: \mathfrak{S}(\kappa_1)$ and $\mathcal{G} \models (\mathfrak{S}(\tau_1, \tau_2)).2 :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow (\mathfrak{S}(\tau_1, \tau_2)).1]$. Then, by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}((\tau_1, \tau_2)) :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$ holds.
- ProjL: $\tau = \tau'.1$ and $\kappa = \kappa_1$. By induction hypothesis (4), we get $\mathcal{G} \models \mathfrak{S}(\tau') :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$. Then by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\tau'.1) :: \mathfrak{S}(\kappa_1)$ holds.
- ProjR: $\tau = \tau'.2$ and $\kappa = \kappa_2[\alpha \leftarrow \tau.1]$. By induction hypothesis (4), we get $\mathcal{G} \models \mathfrak{S}(\tau') :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$. Then by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\tau'.2) :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathfrak{S}(\tau).1]$ holds. Then by Lemma 3.5.31 on page 87(c), we have $\mathcal{G} \models \mathfrak{S}(\tau'.2) :: \mathfrak{S}(\kappa_2[\alpha \leftarrow \tau.1])$.
- Refl: $\kappa = \mathcal{S}(\tau)$. By induction hypothesis (4), $\mathcal{G} \models \mathfrak{S}(\tau) :: \{\star\}$ holds, hence $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau) :: \{\star\}$, hence $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\mathcal{S}(\tau))$ by definition of the logical relation.

- ExtPi: $\kappa = \Pi(\alpha : \kappa_1)\,\kappa_2$. There is a sub-derivation $\Gamma \vdash_{\mathsf{HS}} \kappa_1$ ok, hence by induction hypothesis (1), $\mathcal{G} \models \mathfrak{S}(\kappa_1)$ holds. Let $\mathcal{G}' \sqsupseteq \mathcal{G}$ and assume $\mathcal{G}' \models \mathcal{T}_1 :: \mathfrak{S}(\kappa_1)$. Let $\mathfrak{S}' = \mathfrak{S}, \alpha \mapsto \mathcal{T}_1$. By Lemma 3.5.31 on page 87(c) and Lemma 3.5.36 on page 94 we get $\mathcal{G}' \models \mathfrak{S}' : \Gamma, \alpha :: \kappa_1$. Then by induction hypothesis (4) we get $\mathcal{G}' \models \mathfrak{S}'(\tau\ \alpha) :: \mathfrak{S}'(\kappa_2)$. Hence $\mathcal{G}' \models \mathfrak{S}(\tau)\ \mathcal{T}_1 :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$ since $\alpha \notin \mathrm{fv}(\tau) \cup \mathrm{fv}(\mathfrak{S})$. Hence $\mathcal{G}' \models \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathcal{T}_1]$ holds and then $\mathcal{G} \models \mathfrak{S}(\Pi(\alpha : \kappa_1)\,\kappa_2)$. And finally $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\Pi(\alpha : \kappa_1)\,\kappa_2)$ holds by definition of the logical relation.

- ExtSigma: $\kappa = \Sigma(\alpha : \kappa_1)\,\kappa_2$. By induction hypothesis (1), we have $\mathcal{G} \models \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$. By induction hypothesis (4) we have $\mathcal{G} \models \mathfrak{S}(\tau).1 :: \mathfrak{S}(\kappa_1)$ and $\mathcal{G} \models \mathfrak{S}(\tau).2 :: \mathfrak{S}(\kappa_2[\alpha \leftarrow \tau.1])$. Then by definition of the logical relation, we have $\mathcal{G} \models \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathfrak{S}(\tau).1]$. Then, by Lemma 3.5.32 on page 89 we have $\mathcal{G} \models \mathfrak{S}(\tau).2 :: \mathfrak{S}(\kappa_2)[\alpha \leftarrow \mathfrak{S}(\tau).1]$. Hence by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\Sigma(\alpha : \kappa_1)\,\kappa_2)$ holds.

- Sub: $\kappa = \kappa_2$. By induction hypothesis (4) we get $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\kappa_1)$, and then $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\kappa_2)$ holds by induction hypothesis (3).

Type equivalence judgment:

- EqRefl: Follows from induction hypothesis (4).

- EqSym: Follows from induction hypothesis (5).

- EqTrans: Follows from induction hypothesis (5), from Lemma 3.5.31 on page 87.

- EqLam: Similar to the proof for case Lam.

- EqApp: Similar to the proof for case App.

- EqPair: Similar to the proof for case Pair.

- EqProjL: Similar to the proof for case ProjL.

- EqProjR: Similar to the proof for case ProjR.

- EqExtSingle: $\kappa = \mathcal{S}(\tau')$. By induction hypothesis (4) we have $\mathcal{G} \models \mathfrak{S}(\tau) :: \mathfrak{S}(\mathcal{S}(\tau'))$. Hence by definition of the logical relation, $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') :: \{\star\}$ holds. Then, since $\mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') = \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') \cup \mathfrak{S}(\tau')$, $\mathcal{G} \models \mathfrak{S}(\tau) \cup \mathfrak{S}(\tau') :: \mathfrak{S}(\mathcal{S}(\tau'))$ holds.

- EqExtPi: Similar to the proof for case ExtPi.

- EqExtSigma: Similar to the proof for case ExtSigma.

- EqSub: Follows from induction hypotheses (e) and (c). □

The following lemma is used for the initialization of the logical relation: bindings of wellformed environment are logically related.

Lemma 3.5.38 (Initialization). *If* $\Gamma \vdash_{\mathsf{HS}} \mathrm{ok}$, *then for-all* $\alpha \in \mathrm{dom}\,\Gamma$, $\{\Gamma\} \models \{\alpha\} :: \{\Gamma(\alpha)\}$ *holds. In other words,* $\{\Gamma\} \models \{\mathtt{id}\} : \Gamma$, *where* $\mathtt{id}$ *is the identity substitution.*

Proof. By induction on Γ:

- $\Gamma = \varepsilon$: there is nothing to prove.

- $\underline{\Gamma = \Gamma', \alpha :: \kappa}$: then we have $\Gamma' \vdash_{\mathsf{HS}} \mathrm{ok}$ and $\Gamma' \vdash_{\mathsf{HS}} \kappa\ \mathrm{ok}$ and $\alpha \notin \mathrm{dom}\,\Gamma'$. Then by induction hypothesis, $\{\Gamma'\} \models \{\mathrm{id}\} : \Gamma'$ holds. Hence by Lemma 3.5.30 on page 87, we get $\{\Gamma', \alpha :: \kappa\} \models \{\mathrm{id}\} : \Gamma'$. Then, since $\Gamma' \vdash_{\mathsf{HS}} \kappa\ \mathrm{ok}$, by Lemma 3.5.37 on page 94 we get $\{\Gamma', \alpha :: \kappa\} \models \{\kappa\}$. Then, since $\lceil \eta_\kappa\ \alpha \rceil^{\lceil \Gamma', \alpha :: \kappa \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \alpha \rceil^{\lceil \Gamma', \alpha :: \kappa \rceil}$ by Lemma 3.5.19 on page 82, the Proposition 3.5.35 on page 91(c) entails $\{\Gamma', \alpha :: \kappa\} \models \{\alpha\} :: \{\kappa\}$. Then, by Lemma 3.5.36 on page 94 we get $\{\Gamma', \alpha :: \kappa\} \models \mathrm{id} : \Gamma', \alpha :: \kappa$. □

We can now establish that equivalent kinds (*resp.* types) are logically related.

Proposition 3.5.39. *The following assertions hold:*

- *If* $\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_2$, *then* $\{\Gamma\} \models \{\kappa_1, \kappa_2\}$;
- *If* $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa$, *then* $\{\Gamma\} \models \{\tau_1, \tau_2\} :: \{\kappa\}$.

Proof. Follows from Lemma 3.5.37 on page 94 and Lemma 3.5.38 on the previous page. □

We can eventually conclude with the completeness theorem.

Theorem 3.5.40 (Completeness)**.** *The following assertions hold:*

- *If* $\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_2$, *then* $\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_2 \rceil^{\lceil \Gamma \rceil}$;
- *If* $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa$, *then* $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil \Gamma \rceil}$.

Proof. Follows from Proposition 3.5.39 and Proposition 3.5.35 on page 91. □

3.5.8 Adequacy

The final adequacy theorem ensures that equivalence on wellformed kinds (*resp.* types) is equivalent to convertibility up to insertion of expansors.

Theorem 3.5.41 (Adequacy)**.** *The following assertions hold:*

- $\Gamma \vdash_{\mathsf{HS}} \kappa_1 \equiv \kappa_2$ *holds iff* $\Gamma \vdash_{\mathsf{HS}} \kappa_1\ \mathrm{ok}$ *and* $\Gamma \vdash_{\mathsf{HS}} \kappa_2\ \mathrm{ok}$ *and* $\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_2 \rceil^{\lceil \Gamma \rceil}$ *hold.*
- $\Gamma \vdash_{\mathsf{HS}} \tau_1 \equiv \tau_2 :: \kappa$ *holds iff* $\Gamma \vdash_{\mathsf{HS}} \tau_1 :: \kappa$ *and* $\Gamma \vdash_{\mathsf{HS}} \tau_2 :: \kappa$ *and* $\lceil \eta_\kappa\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau_2 \rceil^{\lceil \Gamma \rceil}$ *hold.*

Proof. Follows from Theorem 3.5.25 on page 84, Theorem 3.5.40, and Proposition 3.1.2 on page 61. □

Thanks to confluence and strong normalization (Theorem 3.5.15 and Theorem 3.5.17 on page 82), this theorem gives a family of algorithms to decide equivalence, defined by the possible strategies to implement normalization or convertibility.

In [Cra07], Crary gives a method to eliminate singleton kinds: in a nutshell, he shows that it suffices to replace every *free* type variable with its η-expansion, then erase the singletons in the type annotations on the arguments of functions, and eventually test for β-equality. Aside from the erasure of singletons inside annotation, this is similar to our method: indeed, complete $\eta_\bullet$-reduction corresponds to η-expansion. The difference relies in the insertion of expansors: we insert expansors at *every* type variable, whenever they are free or bound. We can define a *weak* insertion of expansors, that would perform insertion at free occurrences of variables only, as follows:

$$\textsc{MinVar}\ \frac{}{\Gamma \vdash_{\mathsf{HS}} \alpha \rightsquigarrow \mathcal{S}_{\Gamma(\alpha)}(\alpha)} \qquad \textsc{MinLam}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa_2 \quad \Gamma \vdash_{\mathsf{HS}} \kappa_1\ \mathsf{ok} \quad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash_{\mathsf{HS}} \lambda(\alpha :: \kappa_1)\,\tau \rightsquigarrow \Pi(\alpha : \kappa_1)\,\kappa_2}$$

$$\textsc{MinApp}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \rightsquigarrow \Pi(\alpha : \kappa_2')\,\kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 \rightsquigarrow \kappa_2 \quad \Gamma \vdash_{\mathsf{HS}} \kappa_2 \leqslant \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \tau_1\ \tau_2 \rightsquigarrow \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{MinPair}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \rightsquigarrow \kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 \rightsquigarrow \kappa_2}{\Gamma \vdash_{\mathsf{HS}} (\tau_1, \tau_2) \rightsquigarrow \kappa_1 \times \kappa_2} \qquad \textsc{MinProj}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa_1 \times \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau.\mathrm{i} \rightsquigarrow \kappa_\mathrm{i}}$$

Figure 3.12: Minimal kinds.

Definition 3.5.17 (Weak insertion of expansors)**.**

$$\begin{array}{rcl} \{\varepsilon\} & \triangleq & \varepsilon \\ \{\Gamma, \alpha :: \kappa\} & \triangleq & \{\Gamma\}, \alpha :: \{\kappa\}^{\{\Gamma\}} \end{array}$$

$$\{\alpha\}^\Gamma \triangleq \begin{cases} \eta_{\Gamma(\alpha)}\ \alpha & \text{if } \alpha \in \mathrm{dom}\,\Gamma \\ \alpha & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \{\star\}^\Gamma & \triangleq & \star \\ \{\mathcal{S}(\tau)\}^\Gamma & \triangleq & \mathcal{S}(\{\tau\}^\Gamma) \\ \{\Pi(\alpha : \kappa_1)\,\kappa_2\}^\Gamma & \triangleq & \Pi(\alpha : \{\kappa_1\}^\Gamma)\,\{\kappa_2\}^\Gamma \\ \{\Sigma(\alpha : \kappa_1)\,\kappa_2\}^\Gamma & \triangleq & \Sigma(\alpha : \{\kappa_1\}^\Gamma)\,\{\kappa_2\}^\Gamma \end{array}$$

$$\begin{array}{rcl} \{\eta_\kappa\}^\Gamma & \triangleq & \eta_{\{\kappa\}^\Gamma} \\ \{\lambda(\alpha :: \kappa)\,\tau\}^\Gamma & \triangleq & \lambda(\alpha :: \{\kappa\}^\Gamma)\,\{\tau\}^\Gamma \\ \{\tau_1\ \tau_2\}^\Gamma & \triangleq & \{\tau_1\}^\Gamma\ \{\tau_2\}^\Gamma \\ \{(\tau_1, \tau_2)\}^\Gamma & \triangleq & (\{\tau_1\}^\Gamma, \{\tau_2\}^\Gamma) \\ \{\tau.\mathrm{i}\}^\Gamma & \triangleq & \{\tau\}^\Gamma.\mathrm{i} \end{array}$$

The only difference with insertion (Definition 3.5.9 on page 78) is that the environment is not extended when the insertion function traverses a binder. Therefore, bound variables are left unchanged by weak insertion. Weak insertion has the interesting property that it commutes with substitution, whereas our insertion has a different notion of commutation (Lemma 3.5.22 on page 83). Interestingly, we have checked that the completeness proof continues to work if one uses weak insertion to characterize *type* equivalence; however, strong insertion is still required for the characterization of *kind* equivalence. We show in Section 3.5.9 that our insertion has other interesting properties, that weak insertion does not enjoy.

We think that our approach is more *regular*, because it uses the same procedure for types and kinds, and also more *flexible*, because the commutation between $\beta\pi$ and $\eta_\bullet$ permits to freely interleave reduction of redexes and expansions of expansors. This permits, for instance, to *delay* the unfolding of definitions. However, we insert more expansors, *i.e.* we possibly perform more η-expansions, than Crary's characterization does. This could possibly render more costly an algorithm based on our method.

3.5.9 Insertions of expansors and minimal kinds

In Figure 3.12, we recall the definition of the minimal kind, from [SH06]. Minimal kinds are defined in a similar way to the elimination of subtyping and of strengthening in module systems: variables are given their most precise kind, that is the singleton kind of themselves at the kind given by the environment. Other constructions use their natural rules: the Lam rule for functions, the Pair rule for pairs, *etc*. In the case of applications, we also check that the argument has a lesser kind than the one expected by the left part of the application. Stone and Harper show that this judgment effectively defines minimal kinds:

Lemma 3.5.42 (Soundness of minimal kinds)**.** *If* $\Gamma \vdash_{\mathsf{HS}} \mathrm{ok}$ *and* $\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa$*, then* $\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa$ *holds.*

Lemma 3.5.43 (Completeness of minimal kinds). *If* $\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa$ *and* $\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa'$, *then* $\Gamma \vdash_{\mathsf{HS}} \kappa \leqslant \mathcal{S}_{\kappa'}(\tau)$ *holds.*

Combined with the fact that $\Gamma \vdash_{\mathsf{HS}} \mathcal{S}_{\kappa'}(\tau) \leqslant \kappa'$ (Proposition 3.1.8 on page 61), it shows that the kind κ is indeed minimal.

We can show a lemma that is analogous to Lemma 3.4.22 on page 73: when inserting expansors in a type, adding an additional expansor at its *minimal* kind is superfluous. In other words, normalizing a type up to expansion is the same as normalizing it at its minimal kind.

To prove this result, we first need a preliminary lemma that deals with expansors at a singleton kind: they have the interesting property that they always give the same result, that is, the expansion of the singleton itself.

Lemma 3.5.44. *For any environment* Γ, *types* τ *and* τ', *and kind* κ, *the assertion* $\lceil \eta_{\mathcal{S}_\kappa(\tau')}\ \tau \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau' \rceil^{\lceil\Gamma\rceil}$ *holds.*

Proof. By induction on the size of κ:

- $\kappa = \star$: immediate, since $\mathcal{S}_\star(\tau') = \mathcal{S}(\tau')$.
- $\kappa = \mathcal{S}(\tau'')$: immediate, since $\mathcal{S}_{\mathcal{S}(\tau'')}(\tau') = \mathcal{S}(\tau'')$.
- $\kappa = \Pi(\alpha : \kappa_1)\,\kappa_2$:

$$
\begin{array}{rll}
 & \lceil \eta_{\mathcal{S}_{\Pi(\alpha:\kappa_1)\,\kappa_2}(\tau')}\ \tau \rceil^{\lceil\Gamma\rceil} & \\
= & \lceil \eta_{\Pi(\alpha:\kappa_1)\,\mathcal{S}_{\kappa_2}(\tau'\ \alpha)}\ \tau \rceil^{\lceil\Gamma\rceil} & \\
= & \eta_{\Pi(\alpha:\lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\lceil\mathcal{S}_{\kappa_2}(\tau'\ \alpha)\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil} & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\eta_{\lceil\mathcal{S}_{\kappa_2}(\tau'\ \alpha)\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ (\lceil\tau\rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \alpha)) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ (\lceil\tau'\rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \alpha)) & \text{by induction hypothesis} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\Pi(\alpha:\lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil} & \\
= & \lceil \eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ \tau' \rceil^{\lceil\Gamma\rceil} &
\end{array}
$$

- $\kappa = \Sigma(\alpha : \kappa_1)\,\kappa_2$:

$$
\begin{array}{rll}
 & \lceil \eta_{\mathcal{S}_{\Sigma(\alpha:\kappa_1)\,\kappa_2}(\tau')}\ \tau \rceil^{\lceil\Gamma\rceil} & \\
= & \lceil \eta_{\Sigma(\alpha:\mathcal{S}_{\kappa_1}(\tau'.1))\,\mathcal{S}_{\kappa_2}(\tau'.2)}\ \tau \rceil^{\lceil\Gamma\rceil} & \\
= & \eta_{\Sigma(\alpha:\lceil\mathcal{S}_{\kappa_1}(\tau'.1)\rceil^{\lceil\Gamma\rceil})\,\lceil\mathcal{S}_{\kappa_2}(\tau'.2)\rceil^{\lceil\Gamma,\alpha::\mathcal{S}_{\kappa_1}(\tau'.1)\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil} & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil\mathcal{S}_{\kappa_1}(\tau'.1)\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\mathcal{S}_{\kappa_2}(\tau'.2)\rceil^{\lceil\Gamma,\alpha::\mathcal{S}_{\kappa_1}(\tau'.1)\rceil}[\alpha\leftarrow\lceil\eta_{\mathcal{S}_{\kappa_1}(\tau'.1)}\ \tau.1\rceil^{\lceil\Gamma\rceil}]}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.2) & \\
= & (\eta_{\lceil\mathcal{S}_{\kappa_1}(\tau'.1)\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\mathcal{S}_{\kappa_2}(\tau'.2)\rceil^{\lceil\Gamma\rceil}[\alpha\leftarrow\lceil\eta_{\mathcal{S}_{\kappa_1}(\tau'.1)}\ (\eta_{\mathcal{S}_{\kappa_1}(\tau'.1)}\ \tau.1)\rceil^{\lceil\Gamma\rceil}]}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.2) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil\mathcal{S}_{\kappa_1}(\tau'.1)\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\mathcal{S}_{\kappa_2}(\tau'.2)\rceil^{\lceil\Gamma\rceil}[\alpha\leftarrow\lceil\eta_{\mathcal{S}_{\kappa_1}(\tau'.1)}\ \tau.1\rceil^{\lceil\Gamma\rceil}]}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.2) & \text{by Lemma 3.5.19} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\mathcal{S}_{\kappa_2}(\tau'.2)\rceil^{\lceil\Gamma\rceil}[\alpha\leftarrow\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1]}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.2) & \text{by induction} \\
= & (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\mathcal{S}_{\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \tau'.1]}(\tau'.2)\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau\rceil^{\lceil\Gamma\rceil}.2) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\kappa_2[\alpha\leftarrow\eta_{\kappa_1}\ \tau.'1]\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau\rceil^{\lceil'\rceil}\Gamma.2) & \text{by induction} \\
= & (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}[\alpha\leftarrow\lceil\tau'\rceil^{\lceil\Gamma\rceil}.1]}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.2) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1, \eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}[\alpha\leftarrow\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.1]}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil}.2) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\Sigma(\alpha:\lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ \lceil\tau'\rceil^{\lceil\Gamma\rceil} & \\
= & \lceil \eta_{\Sigma(\alpha:\kappa_1)\,\kappa_2}\ \tau' \rceil^{\lceil\Gamma\rceil} & \square
\end{array}
$$

We now prove the result on minimal kind and convertibility.

Lemma 3.5.45. *If* $\Gamma \vdash_{\mathsf{HS}} \mathsf{ok}$ *and* $\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa$, *then* $\lceil \eta_\kappa\ \tau \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil\Gamma\rceil}$.

Proof. By induction on τ:

- $\underline{\tau = \alpha}$:

$$
\begin{array}{lll}
 & \lceil \eta_{\mathsf{S}_{\Gamma(\alpha)}(\alpha)}\ \alpha \rceil^{\lceil\Gamma\rceil} & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lceil \eta_{\Gamma(\alpha)}\ \alpha \rceil^{\lceil\Gamma\rceil} & \text{by Lemma 3.5.44} \\
= & \eta_{\lceil\kappa\rceil^{\lceil\Gamma_1\rceil}}\ (\eta_{\lceil\kappa\rceil^{\lceil\Gamma_1\rceil}}\ \alpha) & \text{where } \Gamma = \Gamma_1, \alpha :: \kappa, \Gamma_2 \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\lceil\kappa\rceil^{\lceil\Gamma_1\rceil}}\ \alpha & \text{by Lemma 3.5.19} \\
= & \lceil \alpha \rceil^{\lceil\Gamma\rceil} &
\end{array}
$$

- $\underline{\tau = \lambda(\alpha :: \kappa_1)\,\tau_2}$: We have $\Gamma, \alpha{::}\kappa_1 \vdash_{\mathsf{HS}} \tau_2 \rightsquigarrow \kappa_2$ and by induction hypothesis, $\lceil \eta_{\kappa_2}\ \tau_2 \rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_2 \rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}$ (**1**). Then:

$$
\begin{array}{lll}
 & \lceil \eta_{\Pi(\alpha:\kappa_1)\,\kappa_2}\ (\lambda(\alpha :: \kappa_1)\,\tau_2) \rceil^{\lceil\Gamma\rceil} & \\
= & \eta_{\Pi(\alpha:\lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ (\lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\ \lceil\tau_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ ((\lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\ \lceil\tau_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil})\ (\eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \alpha)) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ \lceil\tau_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}[\alpha \leftarrow \eta_{\lceil\kappa_1\rceil^{\lceil\Gamma\rceil}}\ \alpha] & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\,\eta_{\lceil\kappa_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil}}\ \lceil\tau_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil} & \text{by Lemma 3.5.20} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lambda(\alpha :: \lceil\kappa_1\rceil^{\lceil\Gamma\rceil})\ \lceil\tau_2\rceil^{\lceil\Gamma,\alpha::\kappa_1\rceil} & \text{by (1)} \\
= & \lceil \lambda(\alpha :: \kappa_1)\,\tau_2 \rceil^{\lceil\Gamma\rceil} &
\end{array}
$$

- $\underline{\tau = \tau_1\ \tau_2}$: Then, κ is of the form $\kappa_1[\alpha \leftarrow \tau_2]$ (**1**) such that $\Gamma \vdash_{\mathsf{HS}} \tau_1 \rightsquigarrow \Pi(\alpha : \kappa_2')\,\kappa_1$. Moreover, by induction hypothesis, $\lceil \eta_{\Pi(\alpha:\kappa_2')\,\kappa_1}\ \tau_1 \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_1 \rceil^{\lceil\Gamma\rceil}$ (**2**). Moreover, since $\Gamma \vdash_{\mathsf{HS}} \tau_2 :: \kappa_2'$, by Lemma 3.5.42 and rule Sub, it holds that $\Gamma \vdash_{\mathsf{HS}} \tau_2 \equiv (\lambda(\beta : \kappa_2')\,\beta)\ \tau_2 :: \kappa_2'$, and hence $\Gamma \vdash_{\mathsf{HS}} \tau_1\ \tau_2 \equiv \tau_1\ ((\lambda(\beta : \kappa_2')\,\beta)\ \tau_2) :: \kappa_1[\alpha \leftarrow \tau_2]$ holds by EqApp. Then, thanks to adequacy (Theorem 3.5.41 on page 98), we have $\lceil \eta_{\kappa_1[\alpha\leftarrow\tau_2]}\ (\tau_1\ \tau_2) \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_1[\alpha\leftarrow\tau_2]}\ (\tau_1\ ((\lambda(\beta : \kappa_2')\,\beta)\ \tau_2)) \rceil^{\lceil\Gamma\rceil}$ (**3**). Furthermore, it is true that $\Gamma \vdash_{\mathsf{HS}} \kappa_1[\alpha \leftarrow \tau_2] \equiv \kappa_1[\alpha \leftarrow (\lambda(\beta : \kappa_2')\,\beta)\ \tau_2]$, hence by adequacy, $\lceil \kappa_1[\alpha \leftarrow \tau_2] \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \kappa_1[\alpha \leftarrow (\lambda(\beta : \kappa_2')\,\beta)\ \tau_2] \rceil^{\Gamma}$ (**4**) holds. Hence,

$$
\begin{array}{lll}
 & \lceil \tau_1\ \tau_2 \rceil^{\lceil\Gamma\rceil} & \\
= & \lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil} & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\lceil \eta_{\Pi(\alpha:\kappa_2')\,\kappa_1}\ \tau_1 \rceil^{\lceil\Gamma\rceil})\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil} & \text{by (2)} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\Pi(\alpha:\lceil\kappa_2'\rceil^{\lceil\Gamma\rceil})\,\lceil\kappa_1\rceil^{\lceil\Gamma,\alpha::\kappa_2'\rceil}}\ \lceil \tau_1 \rceil^{\lceil\Gamma\rceil})\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil} & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\lceil\kappa_1\rceil^{\lceil\Gamma,\alpha::\kappa_2'\rceil}[\alpha\leftarrow\lceil\eta_{\kappa_2'}\ \tau_2\rceil^{\lceil\Gamma\rceil}]}\ (\lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_2'\rceil^{\lceil\Gamma\rceil}}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil})) & \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\lceil\kappa_1\rceil^{\lceil\Gamma,\alpha::\kappa_2'\rceil}[\alpha\leftarrow\lceil\tau_2\rceil^{\lceil\Gamma\rceil}]}\ (\lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_2'\rceil^{\lceil\Gamma\rceil}}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil})) & \text{by Lemma 3.5.19} \\
= & \eta_{\lceil\kappa_1[\alpha\leftarrow\eta_{\kappa_2'}\ \tau_2]\rceil^{\lceil\Gamma\rceil}}\ (\lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_2'\rceil^{\lceil\Gamma\rceil}}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil})) & \text{by Lemma 3.5.22} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\lceil\kappa_1[\alpha\leftarrow\tau_2]\rceil^{\lceil\Gamma\rceil}}\ (\lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ (\eta_{\lceil\kappa_2'\rceil^{\lceil\Gamma\rceil}}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil})) & \text{by Lemma 3.5.22 and (4)} \\
\overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \eta_{\lceil\kappa_1[\alpha\leftarrow\tau_2]\rceil^{\lceil\Gamma\rceil}}\ (\lceil \tau_1 \rceil^{\lceil\Gamma\rceil}\ \lceil \tau_2 \rceil^{\lceil\Gamma\rceil}) & \text{by (3)} \\
= & \lceil \eta_\kappa\ (\tau_1\ \tau_2) \rceil^{\lceil\Gamma\rceil} &
\end{array}
$$

- $\underline{\tau = (\tau_1, \tau_2)}$: We have $\Gamma \vdash_{\mathsf{HS}} \tau_i \rightsquigarrow \kappa_i$ with $\lceil \eta_{\kappa_i}\ \tau_i \rceil^{\lceil\Gamma\rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_i \rceil^{\lceil\Gamma\rceil}$ (**1**) by induction hypothesis.

$$\textsc{NatVar}\ \frac{}{\Gamma \vdash_{\mathsf{HS}} \alpha \uparrow \Gamma(\alpha)} \qquad \textsc{NatLam}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash_{\mathsf{HS}} \tau \uparrow \kappa_2 \quad \Gamma \vdash_{\mathsf{HS}} \kappa_1\ \mathsf{ok} \quad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash_{\mathsf{HS}} \lambda(\alpha :: \kappa_1)\,\tau \uparrow \Pi(\alpha : \kappa_1)\,\kappa_2}$$

$$\textsc{NatApp}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \uparrow \Pi(\alpha : \kappa_2')\,\kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 \uparrow \kappa_2 \quad \Gamma \vdash_{\mathsf{HS}} \mathcal{S}_{\kappa_2}(\tau_2) \leqslant \kappa_2'}{\Gamma \vdash_{\mathsf{HS}} \tau_1\ \tau_2 \uparrow \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{NatPair}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau_1 \uparrow \kappa_1 \quad \Gamma \vdash_{\mathsf{HS}} \tau_2 \uparrow \kappa_2}{\Gamma \vdash_{\mathsf{HS}} (\tau_1, \tau_2) \uparrow \kappa_1 \times \kappa_2} \qquad \textsc{NatProj}\ \frac{\Gamma \vdash_{\mathsf{HS}} \tau \uparrow \kappa_1 \times \kappa_2}{\Gamma \vdash_{\mathsf{HS}} \tau.\mathrm{i} \uparrow \kappa_i}$$

Figure 3.13: Natural kinds.

Then:

$$\begin{array}{ll} & \lceil \eta_{\kappa_1 \times \kappa_2}\ (\tau_1, \tau_2) \rceil^{\lceil \Gamma \rceil} \\ \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}, \eta_{\lceil \kappa_2 \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}) \\ \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\lceil \tau_1 \rceil^{\lceil \Gamma \rceil}, \lceil \tau_2 \rceil^{\lceil \Gamma \rceil}) \qquad \text{by (1)} \end{array}$$

- $\underline{\tau = \tau_1.\mathrm{i}}$: We have $\Gamma \vdash_{\mathsf{HS}} \tau_1 \rightsquigarrow \kappa_1 \times \kappa_2$ and $\lceil \eta_{\kappa_1 \times \kappa_2}\ \tau_1 \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}$ (**1**) by induction hypothesis. Then:

$$\begin{array}{ll} & \lceil \eta_{\kappa_i}\ \tau_1.\mathrm{i} \rceil^{\lceil \Gamma \rceil} \\ = & \eta_{\lceil \kappa_i \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}.\mathrm{i} \\ \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil \kappa_1 \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}.1, \eta_{\lceil \kappa_2 \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}.2).\mathrm{i} \\ \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & (\eta_{\lceil \kappa_1 \times \kappa_2 \rceil^{\lceil \Gamma \rceil}}\ \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}).\mathrm{i} \\ \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} & \lceil \tau_1 \rceil^{\lceil \Gamma \rceil}.\mathrm{i} \qquad \text{by (1)} \\ = & \lceil \tau_1.\mathrm{i} \rceil^{\lceil \Gamma \rceil} \qquad \square \end{array}$$

Stone and Harper define *natural* kinds for paths (Figure 3.5 on page 62), but we can extend the definition to any type as in Figure 3.13.

There are two differences with the definition of minimal kinds: in the case of variables, one only takes the kind given by the context, without taking its singleton; for the case of applications, the condition on subkinding is slightly changed, by taking the singleton of the argument τ_2 at its natural kind κ_2, instead of κ_2 directly. We can show that natural kinds are valid kinds:

Lemma 3.5.46 (Soundness of natural kind)**.** *If* $\Gamma \vdash_{\mathsf{HS}} \mathsf{ok}$ *and* $\Gamma \vdash_{\mathsf{HS}} \tau \uparrow \kappa$*, then* $\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa$*.*

Proof. By induction on the natural kind judgment. □

Following [SH06], one can show the following relation between natural and minimal kinds:

Lemma 3.5.47. *Assume a type* τ *and an environment* Γ *such that* $\Gamma \vdash_{\mathsf{HS}} \mathsf{ok}$ *holds. Then, there exists a natural kind* κ_n *for* τ (i.e. *such that* $\Gamma \vdash_{\mathsf{HS}} \tau \uparrow \kappa_n$ *holds) iff there exists a minimal kind* κ_m *for* τ (i.e. *such that* $\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa_m$ *holds). Moreover,* $\Gamma \vdash_{\mathsf{HS}} \kappa_m \equiv \mathcal{S}_{\kappa_n}(\tau)$ *holds.*

Proof. We show each implication separately, by induction on judgment in hypothesis. □

This property has an interesting consequence with respect to convertibility:

Lemma 3.5.48. *Assume that* $\Gamma \vdash_{\mathsf{HS}} \mathsf{ok}$*,* $\Gamma \vdash_{\mathsf{HS}} \tau \uparrow \kappa_n$*, and that* $\Gamma \vdash_{\mathsf{HS}} \tau \rightsquigarrow \kappa_m$ *hold. Then the following holds:* $\lceil \eta_{\kappa_n}\ \tau \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_{\kappa_m}\ \tau \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\Gamma}$*.*

Proof. We prove the two parts of convertibility separately. The first part holds thanks to Lemma 3.5.47 on the facing page, Theorem 3.5.41 on page 98, and Lemma 3.5.44 on page 100. The second part is directly proved by Lemma 3.5.45 on page 101. □

This last property is interesting for at least two reasons: first, it shows that minimal and natural kinds are so tightly related notions that convertibility up to insertion cannot distinguish them. Second, it gives an absolute notion of normal form. Stone-Harper's normalization algorithm requires a kind relative to which the types under consideration are normalized. Indeed, the normal forms depend on this kind. We show that it is possible to define normal forms independently of any particular kind, and that the definition obtained is equivalent to choosing either the natural kind or the minimal kind: there is no difference.

3.5.10 A second reading of Stone-Harper's normalization algorithm

Our characterization permits to re-explore the normalization algorithm of from Stone and Harper. We will reuse the definition from Section 3.1.4 on page 62. We can indeed show that every step of the algorithm preserves convertibility up to insertion of expansors. We can show that the following invariants hold for the algorithm:

Lemma 3.5.49 (Invariants of the normalization algorithm)**.** *The following assertions hold:*

Natural kind: *If* $\Gamma \vdash_{\mathsf{HS}} p :: \kappa$ *and* $\Gamma \triangleright p \uparrow \kappa$, *then* $\lceil p \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ p \rceil^{\lceil \Gamma \rceil}$ *holds;*

Head reduction: *If* $\Gamma \vdash_{\mathsf{HS}} \tau :: \star$ *and* $\Gamma \triangleright \tau \rightsquigarrow \tau'$, *then* $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau' \rceil^{\Gamma}$ *holds;*

Head normalization: *If* $\Gamma \vdash_{\mathsf{HS}} \tau :: \star$ *and* $\Gamma \triangleright \tau \Downarrow \tau'$, *then* $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau' \rceil^{\Gamma}$ *holds;*

Type normalization: *If* $\Gamma \vdash_{\mathsf{HS}} \tau :: \kappa$ *and* $\Gamma \triangleright \tau :: \kappa \Longrightarrow \tau'$, *then* $\lceil \eta_\kappa\ \tau \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau' \rceil^{\lceil \Gamma \rceil}$ *holds;*

Path normalization: *If* $\Gamma \vdash_{\mathsf{HS}} p :: \kappa$ *and* $\Gamma \triangleright \tau \longrightarrow \tau' \uparrow \kappa$, *then* $\lceil \tau \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau' \rceil^{\Gamma} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \eta_\kappa\ \tau \rceil^{\Gamma}$ *holds;*

Kind normalization: *If* $\Gamma \vdash_{\mathsf{HS}} \kappa\ \mathsf{ok}$ *and* $\Gamma \triangleright \kappa \Longrightarrow \kappa'$, *then* $\lceil \kappa \rceil^{\Gamma} \longleftrightarrow \lceil \kappa' \rceil^{\Gamma}$ *holds.*

We assume that head reduction and head normalization are performed on types that have a base kind: this is because these two parts of the algorithm always consider such types.

The proof relies on the adequacy theorem (Theorem 3.5.41 on page 98) in several places. This is because we insert expansors at some places where the algorithm does *not* perform η-expansions: it happens every time a substitution is performed by the algorithm. This is the case, for instance, for the invariant on natural kinds. Remark that the proof of Lemma 3.5.45 on page 101 needs the adequacy property as well, and the two proofs are mostly identical.

This is also the case of head reduction, where β-reduction up to insertion of expansors adds an expansor on the argument: β-reduction is translated to $\lceil (\lambda(\alpha : \kappa)\ \tau)\ u \rceil^{\lceil \Gamma \rceil} \xrightarrow{\beta} \lceil \tau \rceil^{\lceil \Gamma \rceil}[\alpha \leftarrow \lceil \eta_\kappa\ u \rceil^{\lceil \Gamma \rceil}]$. However, we know from the adequacy theorem and from Lemma 3.5.22 on page 83 that if $\Gamma \vdash_{\mathsf{HS}} \mathcal{E}[(\lambda(\alpha : \kappa)\ \tau)\ u] \equiv \mathcal{E}[\tau[\alpha \leftarrow u]] :: \star$, then $\lceil (\lambda(\alpha : \kappa)\ \tau)\ u \rceil^{\lceil \Gamma \rceil} \overset{\beta\pi\eta^\bullet}{\longleftrightarrow} \lceil \tau \rceil^{\lceil \Gamma \rceil}[\alpha \leftarrow \lceil u \rceil^{\lceil \Gamma \rceil}]$ holds.

Since our method possibly performs more η-expansions than the original normalization algorithm, our characterization might require more computation. This should be balanced with the flexibility that is brought by our method: the original algorithm maintains the current natural kinds, and threads this kind of information throughout the computation, from one computation to the next. Since some parts of the algorithm demand a kind to further proceed, it does not seem obvious to relax this constraint on the normalization strategy. By contrast, we define reduction without relying on kinds, but rather by internalizing the environment by inserting expansors. As shown by Lemma 3.5.48 on the facing page, this is equivalent to maintaining the natural (or the minimal)

kind. As a consequence, we do not need to keep the natural kind up to date, and, combined with confluence and strong normalization of the reduction relation, it offers a greater flexibility in the choice of the normalization strategy.

We think that the flexibility brought by our method is not only easy to understand, because it uses well-known operations, but can also lead to more efficient algorithms to decide equivalence or normalize types.

3.6 Related work

This chapter is heavily based on Stone and Harpers's system of singleton kinds [SH06], and comparisons have been made throughout the sections. In a nutshell, we developed an alternate manner to decide type and kind equivalence in their system. Our method is based on a confluent and strongly normalizing reduction relation that combines β-reduction and η-expansion, whereas, in the original system, equivalence is decided through the use of an algorithm. As it uses a small-step reduction relation, our method is rather simple to understand, and its properties offer more flexibility to implementers. We believe it could lead to more efficient procedures, by reusing the experience gained by developers of proof checkers for dependent type systems: designing efficient algorithmic tests for convertibility is known to be a subtle art, where clever heuristics play an important role.

Crary [Cra07] developed a sound and complete way to eliminate singleton kinds, that is a transformation that erases singleton annotations, and reduces equivalence to β-equivalence. The more substantial part of the transformation on types consists in replacing free variables by their η-expansion, and performing a head η-expansion. During this operation, η-expansion is kind-directed, and expanding a type τ at the singleton kind $\mathcal{S}(\tau')$ replaces τ with τ': in other words, η-expansion at singleton kinds is analogous to unfolding type definitions. Expansions at other kinds permits to access deep occurrences of singletons, and, consequently, allows to unfold definitions that are not directly available. Our method differs in two points with Crary's: first, our expansions can be considered as *lazy* expansions, since they are defined in a small-step manner, and can be interleaved with other reduction steps. Second, we expand every variables, even bound occurrences.

In the study of dependent type systems, Goguen [Gog05b, Gog05a] showed that $\beta\eta$-equality can be reduced to β-equality, as long as enough η-expansions have been performed: basically, to check for $\beta\eta$-equality, it suffices to η-expand every subterm, and then check for β-equality. As a comparison, we only expand variables instead of every subterm. He sketches an application to a system with singletons, but without subtyping, which, we think, constitutes the main source of difficulty.

Courant [Cou03] defined a small-step reduction relation for a system with singleton kinds, that he proved confluent and strongly normalizing. The corresponding notion of equivalence, however, was intentional. The reduction relation is based on a procedure that is very close to the notion of *natural kinds*. Interestingly, Courant also inserts *marks* $(\tau : \kappa)$ to remember natural kinds, that are close to $\eta_\kappa\ \tau$ in our setting. He also noticed that his marks commute with substitution by keeping marks in substituted variables: this is exactly what happens in our setting.

More recently, Crary [Cra09] defined a normalization procedure based on hereditary substitutions, that is: substitutions that preserve η-long β-normal forms. Although his algorithm looks more involved than Stone and Harper's original system, the proof of completeness is simpler: it does not require the use of complex methods, such as logical relations. He specified and proved his system in Twelf [PS02], and it is now part of the mechanized definition of ML [LCH07]. The original logical relation could not be expressed in Twelf, due to the limitation of Twelf's logic. It is worth remarking that the technique of hereditary substitutions does not lead to an efficient algorithm for testing convertibility, since normal forms have to be computed.

3.7 Future work

We initiated our study of singleton kinds with the intention to extend them to richer types and kinds. More specifically, we wanted to consider a system with records of types classified by dependent record kinds, and width subkinding. Such a system is presented in Chapter 4. Becoming acquainted with the basic system of singleton kinds and its metatheory turned out to be a substantial task that lead to the unanticipated, new results reported in this chapter.

The next step of the plan is to take the characterization of equivalence as a definition for equivalence, and study the obtained system, compare the level of difficulty of its metatheory with the original one, and consider an extension with records. As an immediate benefit, it would simplify the cyclic dependencies between judgments and give a presentation of the system that is closer to dependent type theories, where the conversion rule on types is defined in terms of the reduction relation. The study of the system with the modified definition of equivalence has begun, but its status is too immature to deserve a detailed description.

Modern programming languages offer the possibility to use recursive types whose metatheory is subtle and has been extensively studied. For instance, Stone and Schoonmaker [SS] describe several equational theories with recursive types, some of them including $\beta\eta$-equality. To our knowledge, the interaction between recursive types and singleton kinds has never been studied, although this combination would certainly lead to a clean, powerful and useful type system for designers and implementers of programming languages.

As shorter term goal, we would also like to formalize our development in a proof assistant. The proof of completeness looks like the most challenging part. As already mentioned, we already proved in Coq the Section 3.3 on page 65, that contains the results dealing with the combination of confluence and normalization from commutation diagrams. Another, more practical project, would be to draw a performance-wise comparison between Stone-Harper's original convertibility test, and a well chosen strategy that implements our method.

« Je vis de bonne soupe, et non de beau langage. »

(Molière — Les Femmes savantes)

Chapter 4

A tentative design

In this chapter, we describe the language $\mathsf{F}^{\forall\omega}_{S\leqslant}$ (read F-zip-full), that combines several features so as to serve as a kernel language for modules that is relatively succinct — in particular more succinct than F^{ω}. It extends System F^{ω} in three directions:

- it uses open existential types, whose constructs help to get rid of the block structure of programs that use existential types, and render the definition of existential packages less verbose (see Chapter 2 on page 7);
- it features width and depth subtyping on records as well as on records of types: this way, the user does neither need to manually build and insert coercions to apply a function to an argument, or to instantiate a polymorphic function;
- it incorporates the flexibility of singleton kinds, which model type definitions in programs, along with a powerful notion of equivalence on types.

Consequently, this unique combination of features leverages the use of System F. This choice of features was not made by chance: they all are already present in the ML module language. Indeed, we already advocated that type abstraction in modules are modeled by the constructs of $\mathsf{F}^{\forall}$; the signature inclusion found in ML allows for forgetting value components and type components, as featured by subtyping and subkinding respectively; and finally, type components and type definitions in ML are explained by singleton kinds, as demonstrated in [SH06]. We will see that this combination of features supports programming in a style that is very close to ML modules, but also that $\mathsf{F}^{\forall\omega}_{S\leqslant}$ still lacks some features to provide full parity with ML modules. Unfortunately, it also imports a recurring issue from ML: the *avoidance problem* (Section 4.1.4 on page 115).

In the next sections, we define the language $\mathsf{F}^{\forall\omega}_{S\leqslant}$, then we give some examples and draw comparisons with both System F^{ω} and the ML module language.

4.1 Definitions

In this section, we provide a definition for $\mathsf{F}^{\forall\omega}_{S\leqslant}$: it results from the integration of $\mathsf{F}^{\forall}$ with width subtyping on records and the singleton kind system, which is itself extended to support dependent record kinds (also known as telescopes [Con03]) and width subkinding on records of types. Although the definitions have been carefully designed, no proof on the metatheory of $\mathsf{F}^{\forall\omega}_{S\leqslant}$ has been done yet. We think it would be an interesting challenge to formalize $\mathsf{F}^{\forall\omega}_{S\leqslant}$ and its meta-theory in a proof assistant, especially if one aspires to do so in a modular way. The section ends with the statement of some conjectures about the metatheory of $\mathsf{F}^{\forall\omega}_{S\leqslant}$, namely about natural properties on singletons and type equivalence, and about type safety of the language.

4.1.1 Terms

The syntax of terms is the same as in $\mathsf{F}^{\forall}$, except that both generalization and the constructions for existential types carry the kind at which type variables are introduced. In System $\mathsf{F}^{\forall}$ of Chapter 2, type variables wre restricted to range over the base kind, hence the kind was left implicit.

Definition 4.1.1 (Syntax of terms)**.**

$$\begin{array}{rcl} M & ::= & x \mid \lambda(x:\tau)\,M \mid M\,M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \\ & \mid & \Lambda(\alpha :: \kappa)M \mid M\,\tau \\ & \mid & \{(\ell_i = M_i)^{i\in 1..n}\} \mid M.\ell \\ & \mid & \exists(\alpha :: \kappa)M \mid \mathsf{open}\ \langle\alpha\rangle\ M \\ & \mid & \nu(\alpha :: \kappa)M \mid \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\,M \mid (M:\tau) \end{array}$$

The typing rules of $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$ are given in Figure 4.1 on the facing page. They are similar to the ones of Core $\mathsf{F}^{\forall}$ (Section 2.3 on page 15), augmented with a weakening rule and a subtyping rule. Another minor difference lies in the fact that $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$ features higher order functions on types *à la* F^{ω}, while $\mathsf{F}^{\forall}$ only has the types of System F. We will see in Section 4.2 and in Section 4.3.2 that the rule Weaken is necessary to allow some operations, such as renamings or relocations of existential type variables. We already made a similar remark in Section 2.6.1 on page 36. Notice also that we have to distinguish between two different notions of coercion: an *explicit* coercion $(\cdot:\tau)$ can be inserted between two *coercible* types on the one hand, and an *implicit* coercion can be used between two types in a *subtyping* relation, on the other hand. Coercibility is different from type equivalence (defined as the diagonal of subtyping): type equivalence only propagates equalities that are carried by singleton kinds, while coercibility is larger, because it also propagates equalities provided by equations. Notice that coercibility is already present in $\mathsf{F}^{\forall}$ (see Section 4.1.3 on page 112), but that type equivalence in $\mathsf{F}^{\forall}$ is restricted to syntactic equality.

4.1.2 Types and kinds

Definition 4.1.2 (Syntax of types and kinds)**.**

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \{(\ell_i : \tau_i)^{i\in 1..n}\} \mid \forall(\alpha :: \kappa)\tau \mid \exists(\alpha :: \kappa)\tau \\ & \mid & \lambda(\alpha :: \kappa)\,\tau \mid \tau\,\tau \mid \langle(\ell_i = \tau_i)^{i\in 1..n}\rangle \mid \tau.\ell \\ \kappa & ::= & \star \mid \mathcal{S}(\tau) \mid \Pi(\alpha :: \kappa_1)\,\kappa_2 \mid \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in 1..n}\rangle \end{array}$$

The types of $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$ are those of F^{ω} extended with record types and *records of types*. Their kinds are the same as in the singleton kind system [SH06], extended with dependent records kinds, that classify records of types. We use the usual binding for dependent records: for instance, in the kind $\langle \ell_1\ \mathsf{as}\ \alpha_1 :: \kappa_1\ ;\ \ell_2\ \mathsf{as}\ \alpha_2 :: \kappa_2\ ;\ \ell_3\ \mathsf{as}\ \alpha_3 :: \kappa_3\rangle$, α_1 binds in κ_2 and κ_3, while α_2 binds in κ_3.

In the rest of the chapter, the metavariables R denotes sequences of type fields $(\ell_i : \tau_i)^{i\in I}$, ρ denotes sequences of kind fields $(\ell_i\ \mathsf{as}\ \alpha_i :: \tau_i)^{i\in I}$, and r will denote sequences of term fields $(\ell_i\ \mathsf{as}\ x_i = M_i)^{i\in I}$.

Our judgments for types and kinds differ from [SH06] as follows:

- We allow more types at the base kind, namely: arrows, universal and existential types, and record types. This extension is visible:
  - in the wellformedness judgments in Figure 4.2 on page 111 through the rules WfTypeBaseArrow, WfTypeBaseEmpty, WfTypeBaseRecord, WfTypeForall and WfTypeExists;
  - in the type equivalence judgment in Figure 4.6 on page 113 through the rules EqBaseArrow, EqBaseRecordSwap, EqBaseRecord, EqForall and EqExists;

$$\textsc{Var}\ \frac{\Gamma \vdash \mathsf{ok} \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2 \qquad \Gamma\,\mathrm{pure}}{\Gamma \vdash \lambda(x : \tau_1)\, M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma_1 \curlyvee \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma_2 \vdash M_2 : \tau_2}{\Gamma \vdash M_1\ M_2 : \tau_1}$$

$$\textsc{Let}\ \frac{\Gamma_1 \curlyvee \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2} \qquad \textsc{Proj}\ \frac{j \in 1..n \qquad \Gamma \vdash M : \{(\ell_i : \tau_i)^{i \in 1..n}\}}{\Gamma \vdash M.\ell_j : \tau_j} \qquad \textsc{Empty}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \Gamma\,\mathrm{pure}}{\Gamma \vdash \{\} : \{\}}$$

$$\textsc{Record}\ \frac{\ell_1 \notin \{\ell_i, i \in 2..n\} \qquad \Gamma_1 \curlyvee \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2 \vdash \{(\ell_i = M_i)^{i \in 2..n}\} : \{(\ell_i : \tau_i)^{i \in 2..n}\}}{\Gamma \vdash \{(\ell_i = M_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}}$$

$$\textsc{Gen}\ \frac{\Gamma, \forall\alpha :: \kappa \vdash M : \tau \qquad \Gamma\,\mathrm{pure}}{\Gamma \vdash \Lambda(\alpha :: \kappa) M : \forall(\alpha :: \kappa)\tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash M : \forall(\alpha :: \kappa)\tau'}{\Gamma \vdash M\,\tau : \tau'[\alpha \leftarrow \tau]}$$

$$\textsc{Exists}\ \frac{\Gamma, \exists\alpha :: \kappa \vdash M : \tau}{\Gamma \vdash \exists(\alpha :: \kappa) M : \exists(\alpha :: \kappa)\tau} \qquad \textsc{Open}\ \frac{\alpha \notin \mathrm{dom}\,\Gamma, \Gamma' \qquad \Gamma, \Gamma' \vdash M : \exists(\alpha :: \kappa)\tau}{\Gamma, \exists\alpha :: \kappa, \Gamma' \vdash \mathsf{open}\ \langle\alpha\rangle\ M : \tau} \qquad \textsc{Nu}\ \frac{\alpha \notin \mathrm{ftv}(\tau) \qquad \Gamma, \exists\alpha :: \kappa \vdash M : \tau}{\Gamma \vdash \nu(\alpha :: \kappa) M : \tau}$$

$$\textsc{Sigma}\ \frac{\beta \notin \mathrm{dom}\,\Gamma, \Gamma' \qquad \Gamma, \Gamma', \forall(\alpha = \tau :: \kappa) \vdash M : \tau'}{\Gamma, \exists\beta :: \kappa, \Gamma' \vdash \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M : \tau'} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau \sim \tau' :: \star}{\Gamma \vdash (M : \tau) : \tau} \qquad \textsc{Sub}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \leqslant \tau}{\Gamma \vdash M : \tau}$$

$$\textsc{Weaken}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma' \sqsupseteq \Gamma}{\Gamma' \vdash M : \tau}$$

Figure 4.1: Typing rules of $\mathsf{F}^{\curlyvee\omega}_{\mathsf{S}\leqslant}$.

- We use records of types with dependent record kinds, instead of pairs of types. This is visible
  - in the wellformedness judgment in Figure 4.3 on page 112 and in Figure 4.2 on the next page through the rules WfKindEmpty, WfKindRecord WfTypeEmpty, WfTypeRecord, WfTypeProj and WfTypeExtRecord;
  - in the subkinding judgment in Figure 4.4 on page 112, which is discussed hereafter;
  - in the kind equivalence judgment in Figure 4.5 on page 112, that we directly defined as the diagonal case of subkinding;
  - in the type equivalence judgment in Figure 4.6 on page 113 through the rules EqEmpty, EqRecord, EqProj and EqExtRecord.

From dependent pairs to dependent records

The generalization from dependent pair kinds to dependent record kinds would have been only a cosmetic change, if we had not extended the subkinding relation on record kinds. The addition of fields dropping and fields exchange had unexpected consequences on the definition of the system. Indeed, in a first attempt, we added the following two subkinding rules, that respectively implement dropping of a field and swapping of two fields, and then closed the relation by congruence and transitivity:

$$\textsc{KindSubDrop}\quad \frac{\Gamma \vdash \langle \ell_1 :: \kappa_1 \ ; \ \rho \rangle \ \mathsf{ok}}{\Gamma \vdash \langle \ell_1 :: \kappa_1 \ ; \ \rho \rangle \leqslant \langle \rho \rangle}$$

$$\textsc{KindSubSwap}\quad \frac{\Gamma \vdash \langle \ell_1 \ \mathsf{as} \ \alpha_1 :: \kappa_1 \ ; \ \ell_2 \ \mathsf{as} \ \alpha_2 :: \kappa_2 \ ; \ \rho \rangle \ \mathsf{ok} \qquad \alpha_1 \notin \mathrm{ftv}(\kappa_2) \qquad \alpha_2 \notin \mathrm{ftv}(\kappa_1)}{\Gamma \vdash \langle \ell_1 \ \mathsf{as} \ \alpha_1 :: \kappa_1 \ ; \ \ell_2 \ \mathsf{as} \ \alpha_2 :: \kappa_2 \ ; \ \rho \rangle \leqslant \langle \ell_2 \ \mathsf{as} \ \alpha_2 :: \kappa_2 \ ; \ \ell_1 \ \mathsf{as} \ \alpha_1 :: \kappa_1 \ ; \ \rho \rangle}$$

Unfortunately, this definition does not capture the intention we had: the two kinds $\langle \ell_1 \ \mathsf{as} \ \alpha_1 :: \star \ ; \ \ell_2 :: \mathcal{S}(\alpha_1) \rangle$ and $\langle \ell_2 \ \mathsf{as} \ \alpha_2 :: \star \ ; \ \ell_1 :: \mathcal{S}(\alpha_2) \rangle$ are not comparable in the subkinding relation, while they represent the same piece of information: the projections on the two fields ℓ_1 and ℓ_2 are identical up to equivalence. Indeed, the rule KindSubSwap can never be applied, because the dependency between the two fields cannot be broken. The only way to break this dependency would be to use a name for the whole kind in order to compare the two projections: in other words, one would like to use *extensionality*. Extensionality and subkinding indeed are very closely related: subtyping can generally be encoded as rekinding functions that are $\beta\eta$-equivalent to the identity function. Since extensional rules can already be used in the type wellformedness judgment, and because subkinding is usually semantically characterized by an inclusion of semantics, it naturally comes to mind to use the KindSubExt rule to close the relation under congruence:

$$\textsc{KindSubExt}\quad \frac{\Gamma, \forall \alpha :: \kappa_1 \vdash \alpha :: \kappa_2 \qquad \Gamma \vdash \kappa_2 \ \mathsf{ok}}{\Gamma \vdash \kappa_1 \leqslant \kappa_2}$$

It reads that κ_1 is a subkind of κ_2 if every type of kind κ_1 also has kind κ_2. The kind wellformedness condition on κ_2 ensures that α is not free in κ_2. Then, the extensional rule for records alone is responsible for width subkinding and exchange of fields:

$$\textsc{WfTypeExtRecord}\quad \frac{\Gamma \vdash \tau :: \langle (\ell_i \ \mathsf{as} \ \alpha'_i :: \kappa'_i)^{i \in 1..n} \rangle \qquad I \subseteq 1..n \qquad \Gamma \vdash \langle (\ell_i \ \mathsf{as} \ \alpha_i :: \kappa_i)^{i \in I} \rangle \ \mathsf{ok} \qquad \forall i \in I, \ \Gamma \vdash \tau.\ell_i :: \kappa_i[\alpha_1 \leftarrow \tau.\ell_1] \cdots [\alpha_{i-1} \leftarrow \tau.\ell_{i-1}]}{\Gamma \vdash \tau :: \langle (\ell_i \ \mathsf{as} \ \alpha_i :: \kappa_i)^{i \in I} \rangle}$$

$$\textsc{WfTypeVar}\ \frac{\Gamma \vdash \mathrm{ok} \qquad \Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha :: \kappa} \qquad \textsc{WfTypeBaseArrow}\ \frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma \vdash \tau_2 :: \star}{\Gamma \vdash \tau_1 \to \tau_2 :: \star} \qquad \textsc{WfTypeBaseEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} :: \star}$$

$$\textsc{WfTypeBaseRecord}\ \frac{\Gamma \vdash \tau_1 :: \star \qquad \ell \notin \{\ell_i, i \in I\} \qquad \rho = (\ell_i : \tau_i)^{i\in I} \qquad \Gamma \vdash \{\rho\} :: \star}{\Gamma \vdash \{\ell : \tau\ ;\ \rho\} :: \star}$$

$$\textsc{WfTypeForall}\ \frac{\Gamma, \forall \alpha :: \kappa \vdash \tau :: \star}{\Gamma \vdash \forall(\alpha :: \kappa)\tau :: \star} \qquad \textsc{WfTypeExists}\ \frac{\Gamma, \forall \alpha :: \kappa \vdash \tau :: \star}{\Gamma \vdash \exists(\alpha :: \kappa)\tau :: \star} \qquad \textsc{WfTypeLam}\ \frac{\Gamma, \forall \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash \lambda(\alpha :: \kappa_1)\,\tau :: \Pi(\alpha :: \kappa_1)\,\kappa_2}$$

$$\textsc{WfTypeApp}\ \frac{\Gamma \vdash \tau_1 :: \Pi(\alpha :: \kappa_2)\,\kappa_1 \qquad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 :: \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{WfTypeExtPi}\ \frac{\Gamma, \forall \alpha :: \kappa_1 \vdash \tau\ \alpha :: \kappa_2 \qquad \Gamma \vdash \tau :: \Pi(\alpha :: \kappa_1')\,\kappa_2'}{\Gamma \vdash \tau :: \Pi(\alpha :: \kappa_1)\,\kappa_2} \qquad \textsc{WfTypeEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle :: \langle\rangle}$$

$$\textsc{WfTypeRecord}\ \frac{\Gamma \vdash \langle \ell\ \mathsf{as}\ \alpha :: \kappa\ ;\ \rho\rangle\ \mathrm{ok} \qquad \Gamma \vdash \tau :: \kappa \qquad \Gamma \vdash \langle(\ell_i = \kappa_i)^{i\in I}\rangle :: \langle\rho\rangle[\alpha \leftarrow \tau]}{\Gamma \vdash \langle \ell = \tau\ ;\ (\ell_i = \kappa_i)^{i\in I}\rangle :: \langle \ell\ \mathsf{as}\ \alpha :: \kappa\ ;\ \rho\rangle}$$

$$\textsc{WfTypeProj}\ \frac{j \in 1..n \qquad \Gamma \vdash \tau :: \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in 1..n}\rangle}{\Gamma \vdash \tau.\ell_j :: \kappa_j[\alpha_1 \leftarrow \tau.\ell_1] \cdots [\alpha_{j-1} \leftarrow \tau.\ell_{j-1}]}$$

$$\textsc{WfTypeExtRecord}\ \frac{\Gamma \vdash \tau :: \{(\ell_i\ \mathsf{as}\ \alpha_i' :: \kappa_i')^{i\in 1..n}\} \qquad I \subseteq 1..n \qquad \Gamma \vdash \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in I}\rangle\ \mathrm{ok} \qquad \forall i \in I,\ \Gamma \vdash \tau.\ell_i :: \kappa_i[\alpha_1 \leftarrow \tau.\ell_1] \cdots [\alpha_{i-1} \leftarrow \tau.\ell_{i-1}]}{\Gamma \vdash \tau :: \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in I}\rangle}$$

$$\textsc{WfTypeExtSingle}\ \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \tau :: \mathcal{S}(\tau)} \qquad \textsc{WfTypeSub}\ \frac{\Gamma \vdash \tau :: \kappa' \qquad \Gamma \vdash \kappa' \leqslant \kappa}{\Gamma \vdash \tau :: \kappa}$$

Figure 4.2: Wellformed types.

Notice that we do not require to project the record on all its fields, but only on *some* fields: this is necessary to implement dropping of fields. It obviously implements the possibility to drop some fields, as long as their dependency can be eliminated.

Let us now see how this solves the problem of swapping two dependent fields as above: to prove $\vdash \langle \ell_1\ \mathsf{as}\ \alpha_1 :: \star\ ;\ \ell_2 :: \mathcal{S}(\alpha_1)\rangle \leqslant \langle \ell_2\ \mathsf{as}\ \alpha_2 :: \star\ ;\ \ell_1 :: \mathcal{S}(\alpha_2)\rangle$ it suffices to prove that $\forall \alpha :: \langle \ell_1\ \mathsf{as}\ \alpha_1 :: \star\ ;\ \ell_2\ \mathsf{as}\ :: \mathcal{S}(\alpha_1)\rangle \vdash \alpha :: \langle \ell_2\ \mathsf{as}\ \alpha_2 :: \star\ ;\ \ell_1 :: \mathcal{S}(\alpha_2)\rangle$ holds, thanks to KindSubExt. Then by applying WfTypeExtRecord, it suffices to prove $\forall \alpha :: \langle \ell_1\ \mathsf{as}\ \alpha_1 :: \star\ ;\ \ell_2\ \mathsf{as}\ :: \mathcal{S}(\alpha_1)\rangle \vdash \alpha.\ell_2 :: \star$ and $\forall \alpha :: \langle \ell_1\ \mathsf{as}\ \alpha_1 :: \star\ ;\ \ell_2\ \mathsf{as}\ :: \mathcal{S}(\alpha_1)\rangle \vdash \alpha.\ell_1 :: \mathcal{S}(\alpha.\ell_1)$ hold. The first judgment is easily proved by using the rules WfTypeVar, WfTypeProj and WfTypeSub. The second one is proved using WfTypeVar, WfTypeProj and WfTypeExtSingle.

The rule EqExtRecord is the counterpart of WfTypeExtRecord for type equivalence: it has been changed in a similar way.

Remarks about the empty record kind

One can notice that the empty record kind $\langle\rangle$ is a singleton kind, in the sense that it only contains one equivalence class of types: the equivalence class of the empty record type $\langle\rangle$. Indeed, assume a type τ such that $\Gamma \vdash \tau :: \langle\rangle$. Then, since $\Gamma \vdash \langle\rangle :: \langle\rangle$, we get $\Gamma \vdash \tau \equiv \langle\rangle :: \langle\rangle$ by EqExtRecord. More generally, whenever two types have a record kind, they are equivalent at the empty record kind thanks to the rules WfTypeSub and EqExtRecord.

$$\textsc{WfKindStar}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \star\ \mathrm{ok}} \qquad \textsc{WfKindSingle}\ \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \mathcal{S}(\tau)\ \mathrm{ok}} \qquad \textsc{WfKindPi}\ \frac{\Gamma, \forall \alpha :: \kappa_1 \vdash \kappa_2\ \mathrm{ok}}{\Gamma \vdash \Pi(\alpha :: \kappa_1)\ \kappa_2\ \mathrm{ok}}$$

$$\textsc{WfKindEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle\ \mathrm{ok}} \qquad \textsc{WfKindRecord}\ \frac{\ell \notin \{\ell_i, i \in I\} \qquad \rho = (\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i \in I} \qquad \Gamma, \forall \alpha :: \kappa \vdash \langle \rho \rangle\ \mathrm{ok}}{\Gamma \vdash \langle \ell\ \mathsf{as}\ \alpha :: \kappa\ ;\ \rho \rangle\ \mathrm{ok}}$$

Figure 4.3: Wellformed kinds.

$$\textsc{KindSubForget}\ \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \mathcal{S}(\tau) \leqslant \star} \qquad \textsc{KindSubSingle}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \mathcal{S}(\tau_1) \leqslant \mathcal{S}(\tau_2)} \qquad \textsc{KindSubExt}\ \frac{\Gamma, \forall \alpha :: \kappa_1 \vdash \alpha :: \kappa_2 \qquad \Gamma \vdash \kappa_2\ \mathrm{ok}}{\Gamma \vdash \kappa_1 \leqslant \kappa_2}$$

Figure 4.4: Subkinding.

Singletons at higher kinds

As in [SH06], singleton kinds are restricted to be at the base kind (see rule WfKindSingle). In the same manner as in [SH06], we can extend singletons to higher kinds as follows:

Definition 4.1.3 (Higher-kinded singletons)**.**

$$\begin{aligned}
\mathcal{S}_{\star}(\tau) &\triangleq \mathcal{S}(\tau) \\
\mathcal{S}_{\mathcal{S}(u)}(\tau) &\triangleq \mathcal{S}(\tau) \\
\mathcal{S}_{\Pi(\alpha :: \kappa_1)\ \kappa_2}(\tau) &\triangleq \Pi(\alpha :: \kappa_1)\, \mathcal{S}_{\kappa_2}(\tau\ \alpha) \\
\mathcal{S}_{\langle (\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i \in 1..n} \rangle}(\tau) &\triangleq \langle (\ell_i :: \mathcal{S}_{\kappa_i}(\tau.\ell_i)[\alpha_1 \leftarrow \tau.\ell_1] \cdots [\alpha_{i-1} \leftarrow \tau.\ell_{i-1}])^{i \in 1..n} \rangle
\end{aligned}$$

This definition can be understood as extensionality at the level of kinds.

4.1.3 Coercibility

Two types are *coercible* when they are equivalent in the sense of the singleton kinds system and up to the equations present in the context. The coercibility judgment is indexed by kinds to inherit the full power of type equivalence, which is also indexed by kinds. In the typing rules of Figure 4.1 on page 109, however, coercibility is only used between types at the base kind, *i.e.* the types that classify terms, as coercibility can only be used through the term-level construct of coercion $(\cdot : \tau)$.

One can understand equations as *explicit singletons*, and, conversely, singletons can be seen as *implicit equations*: they both carry a piece of information that connects a type variable to a type, but the way they can be used and their purpose are different. On the one hand, singleton kinds are

$$\textsc{KindEq}\ \frac{\Gamma \vdash \kappa_1 \leqslant \kappa_2 \qquad \Gamma \vdash \kappa_2 \leqslant \kappa_1}{\Gamma \vdash \kappa_1 \equiv \kappa_2}$$

Figure 4.5: Equivalent kinds.

$$\textsc{EqRefl}\ \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau \equiv \tau :: \kappa} \qquad \textsc{EqSym}\ \frac{\Gamma \vdash \tau_2 \equiv \tau_1 :: \kappa}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa} \qquad \textsc{EqTrans}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa \qquad \Gamma \vdash \tau_2 \equiv \tau_3 :: \kappa}{\Gamma \vdash \tau_1 \equiv \tau_3 :: \kappa}$$

$$\textsc{EqBaseArrow}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' :: \star \qquad \Gamma \vdash \tau_2 \equiv \tau_2' :: \star}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' :: \star} \qquad \textsc{EqBaseRecordSwap}\ \frac{\Gamma \vdash \{\ell_1 : \tau_1\ ;\ \ell_2 : \tau_2\ ;\ R\} :: \star}{\Gamma \vdash \{\ell_1 : \tau_1\ ;\ \ell_2 : \tau_2\ ;\ R\} \equiv \{\ell_2 : \tau_2\ ;\ \ell_1 : \tau_1\ ;\ R\} :: \star}$$

$$\textsc{EqBaseEmpty}\ \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \{\} \equiv \{\} :: \star} \qquad \textsc{EqBaseRecord}\ \frac{\ell \notin \{\ell_i, i \in I\} \cup \{\ell_j', j \in J\} \qquad \Gamma \vdash \tau \equiv \tau' :: \star \qquad R = (\ell_i : \tau_i)^{i \in 1..n} \qquad R' = (\ell_i' : \tau_i')^{i \in 1..n'} \qquad \Gamma \vdash \{R\} \equiv \{R'\} :: \star}{\Gamma \vdash \{\ell : \tau\ ;\ R\} \equiv \{\ell : \tau'\ ;\ R'\} :: \star}$$

$$\textsc{EqForall}\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 :: \star \qquad \Gamma, \forall \alpha :: \kappa_1 \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \forall(\alpha :: \kappa_1)\tau_1 \equiv \forall(\alpha :: \kappa_2)\tau_2 :: \star} \qquad \textsc{EqExists}\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 :: \star \qquad \Gamma, \forall \alpha :: \kappa_1 \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \exists(\alpha :: \kappa_1)\tau_1 \equiv \exists(\alpha :: \kappa_2)\tau_2 :: \star}$$

$$\textsc{EqLam}\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \qquad \Gamma, \alpha : \kappa_1 \vdash \tau_1 \equiv \tau_2 :: \kappa'}{\Gamma \vdash \lambda(\alpha :: \kappa_1)\,\tau_1 \equiv \lambda(\alpha :: \kappa_2)\,\tau_2 :: \Pi(\alpha : \kappa_1)\,\kappa'} \qquad \textsc{EqApp}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' :: \Pi(\alpha : \kappa_2)\,\kappa_1 \qquad \Gamma \vdash \tau_2 \equiv \tau_2' :: \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' :: \kappa_1[\alpha \leftarrow \tau_2]}$$

$$\textsc{EqProj}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in 1..n}\rangle \qquad j \in 1..n}{\Gamma \vdash \tau_1.\ell_j \equiv \tau_2.\ell_j :: \kappa_j[\alpha_1 \leftarrow \tau_1.\ell_1] \cdots [\alpha_{j-1} \leftarrow \tau_1.\ell_{j-1}]} \qquad \textsc{EqEmpty}\ \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \langle\rangle \equiv \langle\rangle :: \langle\rangle}$$

$$\textsc{EqRecord}\ \frac{\Gamma \vdash \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in 1..n}\rangle \text{ ok} \qquad \forall i \in 1..n,\ \Gamma \vdash \tau_i \equiv \tau_i' :: \kappa_i[\alpha_1 \leftarrow \tau_1] \cdots [\alpha_{i-1} \leftarrow \tau_{i-1}]}{\Gamma \vdash \langle(\ell_i = \tau_i)^{i \in 1..n}\rangle \equiv \langle(\ell_i = \tau_i')^{i \in 1..n}\rangle :: \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in 1..n}\rangle}$$

$$\textsc{EqExtPi}\ \frac{\Gamma, \alpha : \kappa_1 \vdash \tau_1\ \alpha \equiv \tau_2\ \alpha :: \kappa_2 \qquad \Gamma \vdash \tau_1 :: \Pi(\alpha :: \kappa_1)\,\kappa_3 \qquad \Gamma \vdash \tau_2 :: \Pi(\alpha :: \kappa_1)\,\kappa_4}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \Pi(\alpha : \kappa_1)\,\kappa_2}$$

$$\textsc{EqExtRecord}\ \frac{\Gamma \vdash \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in I}\rangle \text{ ok} \qquad I \subseteq 1..n \qquad \Gamma \vdash \tau_1 :: \langle(\ell_i \text{ as } \alpha_i' :: \kappa_i')^{i \in 1..n}\rangle \qquad \Gamma \vdash \tau_2 :: \langle(\ell_i \text{ as } \alpha_i'' :: \kappa_i'')^{i \in 1..n}\rangle \qquad \forall i \in I,\ \Gamma \vdash \tau_1.\ell_i \equiv \tau_2.\ell_i :: \kappa_i[\alpha_1 \leftarrow \tau_1] \cdots [\alpha_{i-1} \leftarrow \tau_{i-1}]}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in I}\rangle}$$

$$\textsc{EqExtSingle}\ \frac{\Gamma \vdash \tau_1 :: \mathcal{S}(\tau_2)}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \mathcal{S}(\tau_2)}$$

$$\textsc{EqSub}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa_1 \qquad \Gamma \vdash \kappa_1 \leqslant \kappa_2}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa_2}$$

Figure 4.6: Type equivalence.

$$\textsc{CoerceEq}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa}{\Gamma \vdash \tau_1 \sim \tau_2 :: \kappa} \qquad \textsc{CoerceUnfold}\ \frac{\Gamma \vdash \mathrm{ok} \qquad \forall (\alpha = \tau :: \kappa) \in \Gamma}{\Gamma \vdash \alpha \sim \tau :: \kappa} \qquad \textsc{CoerceSym}\ \frac{\Gamma \vdash \tau_2 \sim \tau_1 :: \kappa}{\Gamma \vdash \tau_1 \sim \tau_2 :: \kappa}$$

$$\textsc{CoerceTrans}\ \frac{\Gamma \vdash \tau_1 \sim \tau_2 :: \kappa \qquad \Gamma \vdash \tau_2 \sim \tau_3 :: \kappa}{\Gamma \vdash \tau_1 \sim \tau_3 :: \kappa} \qquad \textsc{CoerceBaseArrow}\ \frac{\Gamma \vdash \tau_1 \sim \tau_1' :: \star \qquad \Gamma \vdash \tau_2 \sim \tau_2' :: \star}{\Gamma \vdash \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' :: \star} \qquad \textsc{CoerceBaseEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\} \sim \{\} :: \star}$$

$$\textsc{CoerceBaseRecord}\ \frac{\Gamma \vdash \tau_1 \sim \tau_1' :: \star \qquad \ell_1 \notin \{\ell_i, i \in I\} \qquad R = (\ell_i : \tau_i)^{i \in I} \qquad R' = (\ell_i : \tau_i')^{i \in I} \qquad \Gamma \vdash \{R\} \sim \{R'\} :: \star}{\Gamma \vdash \{\ell_1 : \tau_1 \ ; \ R\} \sim \{\ell_1 : \tau_1' \ ; \ R'\} :: \star}$$

$$\textsc{CoerceBaseForall}\ \frac{\Gamma \vdash \kappa \sim \kappa' \qquad \Gamma, \forall \alpha :: \kappa \vdash \tau \sim \tau' :: \star}{\Gamma \vdash \forall (\alpha :: \kappa) \tau \sim \forall (\alpha :: \kappa') \tau' :: \star}$$

$$\textsc{CoerceBaseExists}\ \frac{\Gamma \vdash \kappa \sim \kappa' \qquad \Gamma, \forall \alpha :: \kappa \vdash \tau \sim \tau' :: \star}{\Gamma \vdash \exists (\alpha :: \kappa) \tau \sim \exists (\alpha :: \kappa') \tau' :: \star} \qquad \textsc{CoerceLam}\ \frac{\Gamma \vdash \kappa \sim \kappa' \qquad \Gamma, \forall \alpha :: \kappa \vdash \tau \sim \tau' :: \kappa_0}{\Gamma \vdash \lambda (\alpha :: \kappa)\, \tau \sim \lambda (\alpha :: \kappa')\, \tau' :: \Pi (\alpha : \kappa)\, \kappa_0}$$

$$\textsc{CoerceApp}\ \frac{\Gamma \vdash \tau_1 \sim \tau_1' :: \Pi (\alpha : \kappa_2)\, \kappa_1 \qquad \Gamma \vdash \tau_2 \sim \tau_2' :: \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 \sim \tau_1'\ \tau_2' :: \kappa_1 [\alpha \leftarrow \tau_2]} \qquad \textsc{CoerceEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle \sim \langle\rangle :: \langle\rangle}$$

$$\textsc{CoerceRecord}\ \frac{\Gamma \vdash \tau_1 \sim \tau_1' :: \kappa_1 \qquad \ell_1 \notin \{\ell_i, i \in 2..n\} \qquad \Gamma \vdash \langle \ell_1 \text{ as } \alpha :: \kappa_1 \ ; \ \rho \rangle \ \mathrm{ok} \qquad \Gamma \vdash \langle (\ell_i = \tau_i)^{i \in 2..n} \rangle \sim \langle (\ell_i = \tau_i')^{i \in 2..n} \rangle :: \langle \rho \rangle [\alpha \leftarrow \tau_1]}{\Gamma \vdash \langle \ell_1 = \tau_1 \ ; \ (\ell_i = \tau_i)^{i \in 2..n} \rangle \sim \langle \ell_1 = \tau_1' \ ; \ (\ell_i = \tau_i')^{i \in 2..n} \rangle :: \langle \ell_1 \text{ as } \alpha :: \kappa_1 \ ; \ \rho \rangle}$$

$$\textsc{CoerceProj}\ \frac{\Gamma \vdash \tau \sim \tau' :: \langle \ell \text{ as } \alpha :: \kappa \ ; \ \rho \rangle}{\Gamma \vdash \tau.\ell \sim \tau'.\ell :: \kappa} \qquad \textsc{CoerceKind}\ \frac{\Gamma \vdash \tau \sim \tau' :: \kappa_1 \qquad \Gamma \vdash \kappa_1 \sim \kappa_2}{\Gamma \vdash \tau \sim \tau' :: \kappa_2}$$

Figure 4.7: Coercible types.

used to model *definitions at the type level*, hence these definitions must be freely usable once they are introduced in the context. On the other hand, equations are used to *define witnesses* for existential types: coercions are used to specify when witnesses must be hidden (see Chapter 2).

Coercibility is defined in Figure 4.7 and Figure 4.8 on the next page: rules CoerceEq and CoerceUnfold include equivalence and, respectively, unfolding of equations. The other rules are responsible for the transitive, symmetric and contextual closure of the relation.

4.1.4 A powerful notion of subtyping

We define subtyping on types living at the base kind, which correspond to types of terms, in Figure 4.9 on page 116. It is designed to remain as faithful as possible to the judgment of module subtyping, up to Russo's interpretation [Rus03, RRD10], that is:

- subtyping must include the unfolding of type definitions: this is treated by importing the equality judgment on types, that is based on singleton kinds (rule SubEq). This ensures, for instance, that $\Gamma \vdash \alpha \leqslant \tau$ and $\Gamma \vdash \tau \leqslant \alpha$, provided that $\Gamma \vdash \alpha :: \mathcal{S}(\tau)$;
- subtyping must allow the dropping of value and module fields, as well as their permutation;

$$\textsc{KindCoerceEq}\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_2}{\Gamma \vdash \kappa_1 \sim \kappa_2} \qquad \textsc{KindCoerceSym}\ \frac{\Gamma \vdash \kappa_2 \sim \kappa_1}{\Gamma \vdash \kappa_1 \sim \kappa_2} \qquad \textsc{KindCoerceTrans}\ \frac{\Gamma \vdash \kappa_1 \sim \kappa_2 \qquad \Gamma \vdash \kappa_2 \sim \kappa_3}{\Gamma \vdash \kappa_1 \sim \kappa_3} \qquad \textsc{KindCoerceSingle}\ \frac{\Gamma \vdash \tau \sim \tau'}{\Gamma \vdash \mathcal{S}(\tau) \sim \mathcal{S}(\tau')}$$

$$\textsc{KindCoercePi}\ \frac{\Gamma \vdash \kappa_1 \sim \kappa_1' \qquad \Gamma, \forall \alpha :: \kappa_1 \vdash \kappa_2 \sim \kappa_2'}{\Gamma \vdash \Pi(\alpha :: \kappa_1)\, \kappa_2 \sim \Pi(\alpha :: \kappa_1')\, \kappa_2'} \qquad \textsc{KindCoerceEmpty}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle \sim \langle\rangle}$$

$$\textsc{KindCoerceRecord}\ \frac{\Gamma \vdash \kappa \sim \kappa' \qquad \ell \notin \{\ell_i, i \in I\} \qquad \rho = (\ell_i \text{ as } \alpha_i :: \kappa_i)^{i \in I} \qquad \rho' = (\ell_i \text{ as } \alpha_i :: \kappa_i')^{i \in I} \qquad \Gamma, \forall \alpha :: \kappa \vdash \langle \rho \rangle \sim \langle \rho' \rangle}{\Gamma \vdash \langle \ell \text{ as } \alpha :: \kappa \ ;\ \rho \rangle \sim \langle \ell \text{ as } \alpha :: \kappa \ ;\ \rho' \rangle}$$

Figure 4.8: Coercible kinds.

this is implemented by the rules SubDrop and SubSwap;

- subtyping must also permit dropping type and module type fields, as well as permuting them, when possible: this is implemented by importing the subkinding judgment on the bounds of existential and universal types (rules SubExists and SubForall). For example, $\Gamma \vdash \exists(\alpha :: \langle \ell_1 \text{ as } \alpha_1 :: \kappa_1 \ ;\ \ell_2 :: \kappa_2 \rangle)\tau \leqslant \exists(\alpha :: \langle \ell_1 :: \kappa_1 \rangle)\tau$ holds when $\alpha.\ell_2$ does not appear in τ;
- subtyping must allow a concrete type definition to subsume an opaque one: this is again done by using the subkinding judgment on bounds of existential and universal types. For instance, $\Gamma \vdash \exists(\alpha :: \mathcal{S}(\tau'))\tau \leqslant \exists(\alpha :: \star)\tau$ holds, when the two types are wellformed;
- subtyping must be covariant on signatures of structures: that is why it is covariant on record types as well as on the bounds and bodies of existential types (rules SubRecord and SubExists). It was used in the previous examples;
- subtyping must be contravariant on the domain of functors, and covariant on their range: that is why subtyping is contravariant on the bounds and domains of universal types and arrow types, while it is covariant on their bodies (rules SubForall and SubArrow). This way, $\Gamma \vdash \forall(\alpha :: \langle \ell_1 :: \star \rangle)\tau \leqslant \forall(\alpha :: \langle \ell_1 \text{ as } \alpha_1 :: \mathcal{S}(\tau_1) \ ;\ \ell_2 :: \kappa_2 \rangle)\tau$ holds, when the two types are wellformed.

As described by the syntax of $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$ and by its typing rules, type instantiation as well as generalization remain explicit. It is also the case of constructs for open existential types. Notice however, that $\Gamma \vdash \forall(\alpha :: \star)\tau \leqslant \forall(\alpha :: \mathcal{S}(\tau'))\tau$ whenever $\Gamma \vdash \tau' :: \star$ and $\Gamma \vdash \forall(\alpha :: \star)\tau :: \star$ hold. Hence, instantiation within the bound of universals happens implicitly, but the universal quantifier has to be removed in an explicit manner. This is similar to the case of module type subtyping in module systems: functor $(X : \text{sig type } t \text{ end})S$ is a subtype of functor $(X : \text{sig type } t = \tau \text{ end})S$, but a functor of the latter module type has to be explicitly instantiated, even if it can only accept one argument, up to equivalence. In some sense, instantiation is implicit, but destruction of quantifiers is explicit.

The avoidance problem

ML module systems suffer from the *avoidance problem*, a characteristic of the subtyping relation on signatures, which can be stated as follows: « there is generally no principal way to avoid a given type variable ». The avoidance problem is problematic when defining a typechecking algorithm, and it is one of the reasons for the use of *paths* in the ML module system. Notice that it is not a

$$\textsc{SubEq}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \tau_1 \leqslant \tau_2} \qquad \textsc{SubDrop}\ \frac{\Gamma \vdash \{\ell : \tau\ ;\ R\} :: \star}{\Gamma \vdash \{\ell : \tau\ ;\ R\} \leqslant \{R\}} \qquad \textsc{SubSwap}\ \frac{\Gamma \vdash \{\ell_1 : \tau_1\ ;\ \ell_2 : \tau_2\ ;\ R\} :: \star}{\Gamma \vdash \{\ell_1 : \tau_1\ ;\ \ell_2 : \tau_2\ ;\ R\} \leqslant \{\ell_2 : \tau_2\ ;\ \ell_1 : \tau_1\ ;\ R\}}$$

$$\textsc{SubForall}\ \frac{\Gamma \vdash \kappa_2 \leqslant \kappa_1 \qquad \Gamma \vdash \forall(\alpha :: \kappa_1)\tau_1 :: \star \qquad \Gamma, \forall \alpha :: \kappa_2 \vdash \tau_1 \leqslant \tau_2}{\Gamma \vdash \forall(\alpha :: \kappa_1)\tau_1 \leqslant \forall(\alpha :: \kappa_2)\tau_2} \qquad \textsc{SubExists}\ \frac{\Gamma \vdash \kappa_1 \leqslant \kappa_2 \qquad \Gamma \vdash \exists(\alpha :: \kappa_2)\tau_2 :: \star \qquad \Gamma, \forall \alpha :: \kappa_1 \vdash \tau_1 \leqslant \tau_2}{\Gamma \vdash \exists(\alpha :: \kappa_1)\tau_1 \leqslant \exists(\alpha :: \kappa_2)\tau_2}$$

$$\textsc{SubRecord}\ \frac{\ell \notin \operatorname{dom} R \cup \operatorname{dom} R' \qquad \Gamma \vdash \tau \leqslant \tau' \qquad \Gamma \vdash \{R\} \leqslant \{R'\}}{\Gamma \vdash \{\ell : \tau\ ;\ R\} \leqslant \{\ell : \tau'\ ;\ R'\}} \qquad \textsc{SubArrow}\ \frac{\Gamma \vdash \tau_1' \leqslant \tau_1 \qquad \Gamma \vdash \tau_2 \leqslant \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'} \qquad \textsc{SubTrans}\ \frac{\Gamma \vdash \tau_1 \leqslant \tau_2 \qquad \Gamma \vdash \tau_2 \leqslant \tau_3}{\Gamma \vdash \tau_1 \leqslant \tau_3}$$

Figure 4.9: Subtyping rules.

problem that is specific to ML: Ghelli and Pierce [GP92] were also confronted with the avoidance problem when adding existential types to $F_{\leqslant}$.

The problem in $F^{\forall\omega}_{\mathcal{S}\leqslant}$ is the same as in ML. It is present in $F^{\forall\omega}_{\mathcal{S}\leqslant}$ because of the use of the subkinding judgment on the bounds of existential and universal types, which itself suffers from the avoidance problem. A possible "cure" would be to allow subtyping only at coercion sites, application sites and instantiation sites, and allow the free use of type equivalence only: this modification would have the impact of turning the judgmental specification into a more algorithmic definition, that would be very close to the implementation of a typechecker.

The classical example for the avoidance problem, given by [HP05, DCH03], does not work in our setting, because we are not restricted to pairs of types: we can *commute* two fields of records of types.

Example (Classical instance of the avoidance problem)**.** One considers $\kappa = \langle \ell_1 :: \mathcal{S}_{\star\to\star}(\lambda(\beta :: \star)\,\alpha)\ ;\ \ell_2 :: \mathcal{S}(\alpha)\rangle$, *i.e.* a record of types with two concrete fields, in a context where $\alpha :: \star$. The first field is equal to $\lambda(\beta :: \star)\,\alpha$, while the second one equals α. Now consider $\kappa'_\tau = \langle \ell_1 \text{ as } \alpha_1 :: \star \to \star\ ;\ \ell_2 :: \mathcal{S}_\star(\alpha_1\ \tau)\rangle$ for any τ of kind $\star$. The fact is that κ is a subkind of κ'_τ for any well-formed τ of kind $\star$, and there is no kind that is strictly between κ and κ'_τ and avoids the variable α if one disallows swapping of fields. Moreover, κ'_{τ_1} and κ'_{τ_2} are not comparable as long as τ_1 and τ_2 are not equivalent types. Hence one can build an infinity of supertypes that are pairwise incomparable and all minimal. As a consequence, if exchanging fields is forbidden, we have found an instance of the avoidance problem. But in the present case, one can also consider $\kappa'' = \langle \ell_2 \text{ as } \alpha_2 :: \star\ ;\ \ell_1 :: \mathcal{S}_{\star\to\star}(\lambda(\beta :: \star)\,\alpha_2)\rangle$, that precisely strictly lies between κ and κ'_τ for every well-formed τ of kind $\star$, and that avoids the variable α. This kind just expresses the fact that the result of the field ℓ_1 is always equivalent to the field ℓ_2. As a consequence, the above example is not an instance of the avoidance problem in $F^{\forall\omega}_{\mathcal{S}\leqslant}$.

We now give another example, that, we believe, is a correct instance of the avoidance problem in the context of $F^{\forall\omega}_{\mathcal{S}\leqslant}$:

Example (Avoidance problem)**.** One considers $\kappa = \langle \ell_1 :: \mathcal{S}_{\star\to\star}(\lambda(\beta :: \star)\,\alpha)\ ;\ \ell_2 :: \mathcal{S}_{(\star\to\star)\to\star}(\lambda(\gamma :: \star \to \star)\,\alpha)\rangle$ in a context where α has the kind $\star$, *i.e.* a record of types with two concrete fields that define two constant functions, that always yield α as result. Consider now $\kappa'_\tau = \langle \ell_1 \text{ as } \alpha_1 :: \star \to \star\ ;\ \ell_2 :: \mathcal{S}_{(\star\to\star)\to\star}(\lambda(\gamma :: \star \to \star)\,(\alpha_1\ \tau))\rangle$. For any well-formed type τ of kind $\star$, κ is a subkind of κ'_τ and, as in the previous example, κ'_{τ_1} and κ'_{τ_2} are incomparable as long as τ_1 and τ_2 are not equivalent types. We think that there does not exist a kind that strictly lies between κ and all κ'_τ and avoids α, because

$$\textsc{WfEnvEmpty}\ \frac{}{\varepsilon \vdash \mathrm{ok}} \qquad \textsc{WfEnvVar}\ \frac{\Gamma \vdash \tau :: \star \qquad x \notin \mathrm{dom}\,\Gamma}{\Gamma, x : \tau \vdash \mathrm{ok}}$$

$$\textsc{WfEnvForall}\ \frac{\Gamma \vdash \kappa\ \mathrm{ok} \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \forall \alpha :: \kappa \vdash \mathrm{ok}} \qquad \textsc{WfEnvExists}\ \frac{\Gamma \vdash \kappa\ \mathrm{ok} \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \exists \alpha :: \kappa \vdash \mathrm{ok}} \qquad \textsc{WfEnvEq}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \forall (\alpha = \tau :: \kappa) \vdash \mathrm{ok}}$$

Figure 4.10: Wellformed environments.

$$\textsc{EnvWeakRefl}\ \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \sqsupseteq \Gamma} \qquad \textsc{EnvWeakWeaken}\ \frac{\Gamma_1, \Gamma_2 \vdash \mathrm{ok} \qquad \Gamma_1, b, \Gamma_2 \vdash \mathrm{ok} \qquad b \neq \exists \alpha :: \kappa}{\Gamma_1, b, \Gamma_2 \sqsupseteq \Gamma_1, \Gamma_2} \qquad \textsc{EnvWeakTrans}\ \frac{\Gamma_1 \sqsupseteq \Gamma_2 \qquad \Gamma_2 \sqsupseteq \Gamma_3}{\Gamma_1 \sqsupseteq \Gamma_3}$$

Figure 4.11: Environment weakening.

there is no way to express the fact that the two components ℓ_1 and ℓ_2 are functions that yield the same result.

It is important to understand that the avoidance problem has no impact on the metatheory of the system of types and kinds as well as on the kind checking problem: no rule requires a type variable to be avoided at this level. It nevertheless has consequences at the level of terms to perform type checking, since it breaks the minimal-type property: there is indeed one rule that requires a type variable to be avoided, that is the rule Nu. Note that with usual existential types too, the rule Unpack has the same particularity.

4.1.5 Environments

Environment wellformedness is defined in Figure 4.10 as the pointwise extension of type wellformedness, and, as usual, ensures that the bindings are unique, *i.e.* no variable can appear more than once.

Purity of environments is defined as in Chapter 2 on page 7: $\Gamma\,\mathrm{pure}$ holds when it contains no existential bindings.

Weakening of environments is defined in Figure 4.11: essentially, Γ_2 is *weaker* than Γ_1, written $\Gamma_2 \sqsupseteq \Gamma_1$, if it is wellformed and contains more pure bindings than Γ_1. Remember that weakening cannot be proved in general: in Chapter 2 on page 7, we could only prove weakening with bindings that do not depend on existential bindings. This judgment permits weakening even when the binding that is considered depends on an existential binding. We will see in Section 4.2 on page 119 and in Section 4.3.2 on page 122 that it is useful in practice, namely to perform *relocations*, which is an operation that is very often used in our translation from ML to $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$.

Zipping is defined in Figure 4.12 on the next page: it extends the definition used in Core $\mathsf{F}^{\forall}$ (Section 2.3 on page 15) with kinded bindings. As in Core $\mathsf{F}^{\forall}$, zipping is *not* symmetric, due to the last line of the figure.

$$
\begin{array}{rcll}
\varepsilon \curlyvee \varepsilon & = & \varepsilon & \\
\Gamma_1, x : \tau \curlyvee \Gamma_2, x : \tau & = & (\Gamma_1 \curlyvee \Gamma_2), x : \tau & \text{if } x \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall \alpha :: \kappa \curlyvee \Gamma_2, \forall \alpha :: \kappa & = & (\Gamma_1 \curlyvee \Gamma_2), \forall \alpha :: \kappa & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \forall (\alpha = \tau :: \kappa) \curlyvee \Gamma_2, \forall (\alpha = \tau :: \kappa) & = & (\Gamma_1 \curlyvee \Gamma_2), \forall (\alpha = \tau :: \kappa) & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha :: \kappa \curlyvee \Gamma_2 & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha :: \kappa & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1 \curlyvee \Gamma_2, \exists \alpha :: \kappa & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha :: \kappa & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2 \\
\Gamma_1, \exists \alpha :: \kappa \curlyvee \Gamma_2, \forall \alpha :: \kappa & = & (\Gamma_1 \curlyvee \Gamma_2), \exists \alpha :: \kappa & \text{if } \alpha \notin \operatorname{dom} \Gamma_1, \Gamma_2
\end{array}
$$

Figure 4.12: Zipping environments.

4.1.6 Dynamic semantics

The dynamic semantics is exactly the same as the one of Core $\mathsf{F}^{\curlyvee}$ (Figure 2.12 on page 22), with the extended syntax, *i.e.* kind annotations for type variables. We do not repeat its definition in this chapter.

4.1.7 Conjectures

Although we have not proved any result about $\mathsf{F}^{\curlyvee\omega}_{\mathcal{S}\leqslant}$, we conjecture that higher-kinded singletons faithfully encode singletons at higher kinds, as it is the case in [SH06]. More specifically, we conjecture the following properties:

Conjecture 4.1.1 (Properties of higher kinded singletons)**.**

Singleton introduction: *if* $\Gamma \vdash \tau :: \kappa$, *then* $\Gamma \vdash \tau :: \mathcal{S}_\kappa(\tau)$ *holds;*

Singleton elimination: *if* $\Gamma \vdash \tau :: \mathcal{S}_\kappa(\tau')$ *and* $\Gamma \vdash \tau' :: \kappa$, *then* $\Gamma \vdash \tau \equiv \tau' :: \mathcal{S}_\kappa(\tau')$ *holds;*

Singleton forgetting: *if* $\Gamma \vdash \tau :: \kappa$ *then* $\Gamma \vdash \mathcal{S}_\kappa(\tau) \leqslant \kappa$ *holds;*

Compatibility with equivalence: *if* $\Gamma \vdash \tau \equiv \tau' :: \kappa$ *and* $\Gamma \vdash \kappa \equiv \kappa'$, *then* $\Gamma \vdash \mathcal{S}_\kappa(\tau) \equiv \mathcal{S}_{\kappa'}(\tau')$ *holds;*

Compatibility with subkinding: *if* $\Gamma \vdash \tau :: \kappa_1$ *and* $\Gamma \vdash \kappa_1 \leqslant \kappa_2$, *then* $\Gamma \vdash \mathcal{S}_{\kappa_1}(\tau) \leqslant \mathcal{S}_{\kappa_2}(\tau)$ *holds.*

We also conjecture that the minimal kind algorithm can be extended to $\mathsf{F}^{\curlyvee\omega}_{\mathcal{S}\leqslant}$ in a straightforward way and proved correct and complete. We guess the same can be done for the normalization algorithm, although it is likely that extending the completeness proof would be difficult, since it is already the hardest part of [SH06] and of Chapter 3.

Conjecture 4.1.2 (Consistency of equivalence)**.** *Type equality is consistent, in the sense that it cannot equate two distinct constructors.*

Notice, however, that two records of types of different arity can be equivalent, as exemplified on page ??.

Based on Conjecture 4.1.2, we also surmise that the coercibility is consistent:

Conjecture 4.1.3. *Type coercibility is consistent, in the sense that it cannot equate two distinct constructors.*

Under these assumptions, the proof of soundness for Core $\mathsf{F}^{\curlyvee}$ should be easily extended, since the only major difference is on the language of types. The presence of subtyping is another difference, but we guess it should not raise difficult problems.

Conjecture 4.1.4. $\mathsf{F}^{\curlyvee\omega}_{\mathcal{S}\leqslant}$ *enjoys the subject reduction and progress properties.*

We have not explored any of the above conjectures yet.

4.2 Examples and remarks

In this section, we give several examples of uses of $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$, along with their corresponding instances in ML: we focus on local type definitions, and relocations of type components.

4.2.1 Local definitions

Previous work on singleton kinds encode local type definitions as follows:

$$\mathsf{def}\ \alpha :: \kappa = \tau\ \mathsf{in}\ M \triangleq (\lambda(\alpha :: \mathcal{S}_\kappa(\tau))\, M)\, \tau$$

As M is parametrized by a singleton kind, M can exploits the fact that α equals τ at the kind κ. Then, the same type τ is given as argument to restore the type of M.

While perfect valid, the module equivalent of this encoding is unnatural: to provide the definition $\mathsf{type}\ t = \tau$ to the module M, one could also use the same trick, that is, write:

$$(\mathsf{functor}\ (X : \mathsf{sig}\ \mathsf{type}\ t = \tau\ \mathsf{end})\mathsf{open}\ \langle X\rangle\ <\text{definition of M}>)\ (\mathsf{struct}\ \mathsf{type}\ t = \tau\ \mathsf{end})$$

But this might seem a bit artificial or convoluted. Instead, one generally prefers to write something like

$$\mathsf{type}\ t = \tau <\text{definition of M}>$$

but then the type component t is exported. Bearing in mind Russo's interpretation, this corresponds to an existential package, which we would write as follows, if the pack construct permitted to bind a variable to a definition:

$$\mathsf{unpack}\ (\mathsf{pack}\ \langle \alpha = \tau, M\rangle\ \mathsf{as}\ \exists(\alpha :: \mathcal{S}_\kappa(\tau))\tau')\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ x$$

The existential package is immediately destructed to restore the original type of M. Notice that the non-escaping condition can always be satisfied, since α can always be replaced by its definition by equivalence.

In $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$, a natural way to express local definitions is:

$$\mathsf{def}\ \alpha :: \kappa = \tau\ \mathsf{in}\ M \triangleq \nu(\alpha :: \mathcal{S}_\kappa(\tau))\Sigma\ \langle\alpha\rangle\ (\alpha = \tau)\, M$$

That is, we create an open existential type, which is immediately restricted. Notice that, as opposed to the other solutions we described, the *structure* of the type of M is kept unchanged: what varies is the typing context only. The first solution we gave would also work in $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$, but its use would be too restrictive because of the purity condition on the body of generalized terms (rule Gen in Figure 4.1).

4.2.2 Renaming or relocation of existential items

Assume that a term M exports two type components β_1 and β_2 of kinds κ_1 and κ_2, respectively, that is, M has type τ in an environment that contains the bindings $\exists\beta_1 :: \kappa_1$ and $\exists\beta_2 :: \kappa_2$. One may want to expose these two type components in a different way to the rest of the program, without changing M itself: for instance one would like to present them as a pair of types. This can be done as follows in $\mathsf{F}^{\forall\omega}_{\mathcal{S}\leqslant}$:

$$\begin{array}{l}\nu(\beta_1 :: \kappa_1)\nu(\beta_2 :: \kappa_2)\\ \quad \Sigma\ \langle\beta\rangle\ (\beta = \langle \ell_1 = \beta_1\ ;\ \ell_2 = \beta_2\rangle)\\ \qquad (M : \tau[\beta_1 \leftarrow \beta.\ell_1][\beta_2 \leftarrow \beta.\ell_2])\end{array}$$

Basically, one defines β as the pair of β_1 and β_2, then applies a coercion to use β instead of the other two type components, and, eventually, hides β_1 and β_2 by requiring them to be local. It is important to notice that this example would not be welltyped without the WEAKEN rule: indeed, the Σ we introduce inserts an equation in the typing context, which depends on β_1 and β_2, that has to be removed by weakening to allow the use of open or Σ on β_1 or β_2 in the definition of M. This technique of relocation is heavily used in our translation from ML to $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$ in Section 4.3.2 on page 122. We already introduced it in Section 2.6.1 on page 36, though in a more limited setting.

We now describe the counterpart of the above example in ML. Assume that the module A has for signature sig type t_1 type t_2 D end where t_1 and t_2 are type components (concrete or abstract), and D is the rest of the signature. The goal is to *relocate* the two type components of A into a submodule C. One proceeds as follows:

$$
\begin{array}{l}
\mathsf{module}\ A' : \mathsf{sig} \\
\quad \mathsf{module}\ C = \mathsf{struct} \\
\quad\quad \mathsf{type}\ t_1 \quad \mathsf{type}\ t_2 \\
\quad \mathsf{end} \\
\quad D[t_1 \leftarrow C.t_1][t_2 \leftarrow C.t_2] \\
\mathsf{end} = \mathsf{struct} \\
\quad \mathsf{module}\ C = \mathsf{struct} \\
\quad\quad \mathsf{type}\ t_1 = A.t_1 \quad \mathsf{type}\ t_2 = A.t_2 \\
\quad \mathsf{end} \\
\quad \mathsf{include}\ A \\
\mathsf{end}
\end{array}
$$

This piece of code defines one submodule C that redefines the two type components to relocate. Then, a signature ascription makes manifest the use of this submodule in the signature, and hides the type components t_1 and t_2 and the definitions of $C.t_1$ and $C.t_2$ at the same time. The parallel with the piece of code from $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$ is obvious; however a substantial difference remains: since terms and types are interleaved in ML, while they are strongly separated in $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$, type components must be repeated in ML in the definition of A' *and* in the signature ascription, whereas it is only done once in the piece of code from $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$.

4.2.3 Phase-split style

Programming in $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$ imposes a *phase-split style*, that is to say, type components and value components are separated. For instance, the module expression

$$
\begin{array}{l}
\mathsf{module}\ M = \mathsf{struct} \\
\quad \mathsf{type}\ t = \mathit{bool} \quad \mathsf{let}\ a = \mathsf{true} \\
\quad \mathsf{module}\ N = \mathsf{struct} \\
\quad\quad \mathsf{type}\ u = \mathit{int} \quad \mathsf{let}\ b = 12 \\
\quad \mathsf{end} \\
\mathsf{end}
\end{array}
$$

would be represented in $\mathsf{F}^{\Upsilon\omega}_{\mathcal{S}\leqslant}$ by the term

$$
\begin{array}{l}
\Sigma\ \langle\beta\rangle\ (\alpha = \langle t = \mathit{bool}\ ;\ N = \langle u = \mathit{int}\rangle\rangle :: \langle t :: \mathcal{S}(\mathit{bool})\ ;\ \langle N :: \langle u :: \mathcal{S}(\mathit{int})\rangle\rangle\rangle \\
\quad \{a = \mathsf{true}\ ;\ N = \{b = 12\}\}
\end{array}
$$

This term defines a transparent static part β of kind $\langle t :: \mathcal{S}(\mathit{bool})\ ;\ \langle N :: \langle u :: \mathcal{S}(\mathit{int})\rangle\rangle\rangle$ on the one hand, and a dynamic part $\{a = \mathsf{true}\ ;\ N = \{b = 12\}\}$. One also has to give the *witness* of the static part, which is $\langle t = \mathit{bool}\ ;\ N = \langle u = \mathit{int}\rangle\rangle$. The structure of the initial structure is split in two parts:

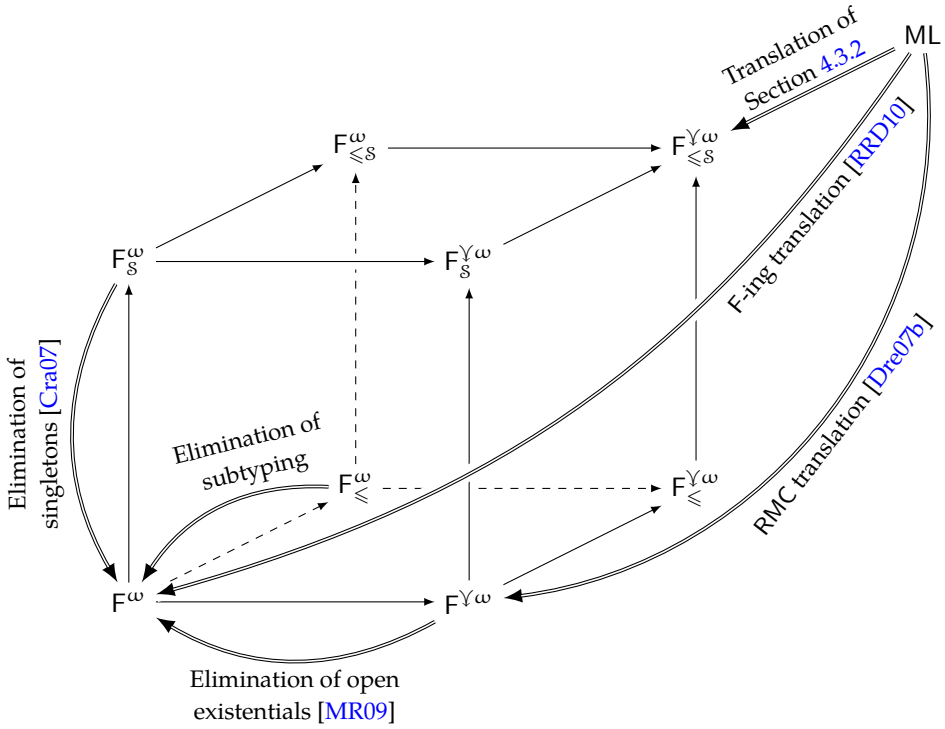


Figure 4.13: Extending System F^{ω} in three directions, and translating back to System F.

the record of type components and the record of value components. The structure of the static part is repeated twice to specify both the witness of the open existential and its kind.

The signatures have to be split in a similar manner.

4.3 Comparisons

In this section, we compare the drawbacks and benefits of $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ with respect to System F^{ω} and then to the ML module system. We will base our comparisons on characteristics of the different translations that are drawn as double arrows in Figure 4.13. For this purpose, a new translation from ML to $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ is sketched in Section 4.3.2 on the next page.

4.3.1 $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ vs. System F^{ω}

$\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ is a conservative extension of F^{ω}, and this extension spreads in three directions, that correspond to the dimensions of the cube of Figure 4.13:

- $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ features the open existential types of $\mathsf{F}^{\curlyvee}$, that allow more freedom in the structure of the program when using existential types, as shown by the elimination of open existential types in Section 2.5.2 on page 30;
- The language of types used by F^{ω} is the simply-typed λ-calculus equipped with extensional equality, whereas $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ uses the simply typed λ-calculus with singleton kinds, which permits to model type definitions, and hence enables more compact code to be written, by factoring

the definition of types: Crary [Cra07] eliminates singleton kinds by replacing type variables by their η-expansions at their natural kind, where expanding at a singleton kind corresponds to unfolding the definition represented by the singleton. This is also the approach we took in Chapter 3, but in a more incremental and demand-driven style;

- $\mathsf{F}^{\forall\omega}_{S\leqslant}$ also adds *subtyping* to F^{ω} by means of width subtyping on records and also on bounds of existential and universal types: there is width subkinding on records of types and singleton kinds can be forgotten. Moreover, subtyping and subkinding are closed under the equivalence induced by the singleton kinds system. This frees the programmer from manually inserting coercions, that would normally be inserted by a translation that eliminates subtyping.

System F^{ω} and $\mathsf{F}^{\forall\omega}_{S\leqslant}$ share some design aspects: management of universal types (generalization and instantiation) and of existential types (package creation and opening) remain *explicit* in both languages. It is fundamental to notice that $\mathsf{F}^{\forall\omega}_{S\leqslant}$ gathers the conveniences brought by open existential types, singleton kinds and subtyping, which render it more convenient to program with and less verbose than F^{ω}.

4.3.2 $\mathsf{F}^{\forall\omega}_{S\leqslant}$ vs. ML

The previous section, which gave a comparison between System F^{ω} and $\mathsf{F}^{\forall\omega}_{S\leqslant}$, showed the width of the gap that separates them in terms of concision and structure of programs. The interpretation of ML into F^{ω} [RRD10] also highlighted the superiority of ML modules over F^{ω} in terms of concision. In this section, we draw a comparison between ML and $\mathsf{F}^{\forall\omega}_{S\leqslant}$ by sketching and studying a translation from ML to $\mathsf{F}^{\forall\omega}_{S\leqslant}$. We will see that the two languages differ only on a few points, which raises the possibility of directly programming with $\mathsf{F}^{\forall\omega}_{S\leqslant}$.

Sketch of a translation from ML to $\mathsf{F}^{\forall\omega}_{S\leqslant}$

Our translation is heavily based on F-ing [RRD10] and borrows ideas from [DCH03, LCH07, Dre07a, Sto05]. As in F-ing [RRD10], it is parametrized by the base language. It improves on F-ing in the following ways:

- it makes use of singleton kinds to prevent unfolding type definitions;
- it makes use of open existential types to keep the structure of the generated code as close as possible to the structure of the original one. The same property was achieved by the translation into RTG [Dre07a];
- thanks to singleton kinds, type components are treated in a uniform manner, regardless whether they are transparent or opaque, as suggested in [Sto05];
- it makes use of records of types to enforce the invariant that each module has exactly one type component: this enables the *phase separated* style at every level of the program (see Section 4.2.3 on page 120);
- the translation benefits from our subtyping relation and avoids the insertion of function coercions that witness occurrences of subtyping.

The source languages for modules, signatures, structure declarations and signature declarations is given in Figure 4.14 on the next page. The kinds k used in this module language are a subset of the kinds κ of $\mathsf{F}^{\forall\omega}_{S\leqslant}$ that were previously defined in Section 4.1.2 on page 108. In the translation judgments, the metavariable R denotes fields of record types, ρ denotes fields of record kinds and r

$$
\begin{array}{rcll}
A & ::= & X & \text{(Modules)} \\
& | & \mathsf{functor}\ (X : S) A \quad | \quad A\ A & \\
& | & \mathsf{struct}\ B\ \mathsf{end} \quad | \quad A.X & \\
& | & (A : S) & \\
\\
B & ::= & \varepsilon & \text{(Structure bindings)} \\
& | & \mathsf{type}\ X :: k = u\ ;\ B & \\
& | & \mathsf{val}\ X = e\ ;\ B & \\
& | & \mathsf{module}\ X = A\ ;\ B & \\
& | & \mathsf{module\ type}\ X = S\ ;\ B & \\
& | & \mathsf{include}\ A\ ;\ B & \\
\\
k & ::= & \star \quad | \quad \kappa \to \kappa & \text{(Kinds)}
\end{array}
\qquad
\begin{array}{rcll}
S & ::= & X & \text{(Signatures)} \\
& | & \mathsf{functor}\ (X : S) S & \\
& | & A.X & \\
& | & \mathsf{sig}\ D\ \mathsf{end} \quad | \quad S\ \mathsf{with\ type}\ X :: k = u & \\
\\
D & ::= & \varepsilon & \text{(Signature bindings)} \\
& | & \mathsf{type}\ X :: k = u\ ;\ D \quad | \quad \mathsf{type}\ X :: k\ ;\ D & \\
& | & \mathsf{val}\ X : u\ ;\ D & \\
& | & \mathsf{module}\ X : S\ ;\ D & \\
& | & \mathsf{module\ type}\ X = S\ ;\ D & \\
& | & \mathsf{include}\ S\ ;\ D &
\end{array}
$$

Figure 4.14: Syntax of an idealized ML module language.

denotes fields of records, as previously introduced. We also introduce a class of contexts b, defined as follows:

$$b \quad ::= \quad [\cdot] \quad | \quad \mathsf{let}\ x = M\ \mathsf{in}\ b \quad | \quad \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,b$$

These contexts are used to build records with internal binding. Since they are not primitive in $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$, we encode them using let-bindings. For instance, the ML structure

```
struct
  let a = 1
  let b = a + 42
end
```

will be expressed as the term

$$\mathsf{let}\ x_a = 1\ \mathsf{in}\ \mathsf{let}\ x_b = x_a + 42\ \mathsf{in}\ \{\ell_a = x_a\ ;\ \ell_b = x_b\}$$

that is equal to $b[\{\ell_a = x_a\ ;\ \ell_b = x_b\}]$ where b is the context $\mathsf{let}\ x_a = 1\ \mathsf{in}\ \mathsf{let}\ x_b = x_a + 42\ \mathsf{in}\ [\cdot]$. In fact, the translation builds the context and the record separately, and assemble them, once they are both constructed.

We also make use of two operators on fields, that is, on association lists, that help to construct records:

Disjoint union: $m_1 \otimes m_2$ is list concatenation, under the condition that m_1 and m_2 have disjoint domains. Otherwise, it fails.

Right-leaning union: $m_1 \Subset m_2$ is list concatenation with shadowing, that is bindings in m_2 are preferred over those in m_1. If $m \setminus L$ denotes the association list m where the bindings that occur in the list L have been removed, then the equation $m_1 \Subset m_2 = (m_1 \setminus \mathrm{dom}\, m_2) \otimes m_2$ holds.

The translation from modules to $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ is composed of six judgments, that we describe one after the other.

Translation of terms of the base language

The judgment $\Gamma \vdash e \Longmapsto M : \tau$ reads « in the environment Γ, the term e translates to M of type τ ». The translation is parametric with respect to this judgment. We assume that the translation of terms of the base language produces welltyped terms, *i.e.*: if $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash e \Longmapsto M : \tau$, then $\Gamma \vdash M : \tau$ holds.

Translation of types of the base language

The judgment $\Gamma \vdash u \Longmapsto \tau :: \kappa$ reads « in the environment Γ, the type u translates to τ of kind κ ». The translation is parametric with respect to this judgment. Similarly, we assume that the translation of types of the base language produces wellkinded types, *i.e.*: if $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash u \Longmapsto \tau :: \kappa$, then $\Gamma \vdash \tau :: \kappa$ holds.

Translation of modules (Figure 4.15 on the next page)

The judgment $\Gamma \vdash A \Longmapsto \alpha :: \kappa \triangleright M : \tau$ reads « in the environment Γ, the module A translates to its static part α (*i.e.* its type components), of kind κ and its dynamic part M (*i.e.* its value components) of type τ ». The following invariant about wellformedness should be kept in mind when reading the rules: if $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash A \Longmapsto \alpha :: \kappa \triangleright M : \tau$ then $\Gamma, \exists \alpha :: \kappa \vdash M : \tau$ holds. A detailed description of each rule follows.

Rule ModVar To translate a module variable X, one looks up in the environment its static part α_X of kind κ and its dynamic part x_X of type τ. One produces a term that has α_X as static part and x_X as dynamic part, that is $\Sigma\,\langle\beta\rangle\,(\beta = \alpha_X)\,x_X$, where β has kind $\mathcal{S}_\kappa(\alpha_X)$, so that β is equivalent to α_X. The singleton kind used here is reminiscent of Lillibridges's selfification [Lil97] or Leroy's [Ler96] strengthening.

Rule ModLam A functor translates to a polymorphic function returning a term of an existential type. Hence, we close the static part of the body into an existential, and then parametrize it by the static part and the dynamic part of the argument. Since a functor does not export type components, its static part is empty, *i.e.* it has the empty record kind.

Rule ModApp To translate a module application $A_1\ A_2$, one has to instantiate the left side M_1 of the application with the static part and then with the dynamic part of the right side M_2 of the application. But in ML, an application only exports the type components of the result of the functor that is applied. Consequently, we hide (*i.e.* restrict) the static parts of M_1 and M_2 and open the result of the application to release its static part in the environment. One checks that the functor is given an argument that has a lesser type than the expected one, so that the application is welltyped. One also checks that one can find a result type for the instantiated functor that does not depend on the static parts whose scopes are restricted. Notice that this last check as well as the two scope restrictions can fail: however, in the case of ML modules, applications are restricted to *paths* (a condition that we do not enforce), so that the static parts always have singleton kinds and, consequently, the three checks cannot fail. In ML, the signatures of paths are always *transparent*, since their type components have been strengthened when typing the variables at the root of the paths.

Rule ModStruct To translate a structure struct B end, one translates its bindings B. This gives a list of static parts, a list of declarations b and a list of record fields r that have the types described by R. One builds the records resulting from inserting the record $\{r\}$ into the context b. Then, one *relocates* the static parts into a single, compound, static part.

Rule ModProj A module projection simply translates to a projection whose static part is relocated. One checks that the relocation is possible, that is to say, one requires a type for the projection

$$\textsc{ModVar}\quad \frac{\alpha_X :: \kappa, x_X : \tau \in \Gamma \qquad \beta \text{ is fresh}}{\Gamma \vdash X \Longmapsto \beta :: \mathcal{S}_\kappa(\alpha_X) \triangleright \Sigma\,\langle\beta\rangle\,(\beta = \alpha_X)\,x_X : \tau}$$

$$\textsc{ModLam}\quad \frac{\Gamma \vdash S \Longmapsto \exists(\alpha_X :: \kappa')\tau' \qquad \Gamma, \alpha_X :: \kappa', x_X : \tau' \vdash A \Longmapsto \beta :: \kappa \triangleright M : \tau \qquad \alpha_X, x_X, \gamma \sharp \operatorname{dom}\Gamma}{\begin{array}{l}\Gamma \vdash \mathsf{functor}\ (X : S)A \Longmapsto \\ \gamma :: \langle\rangle \triangleright \Sigma\,\langle\gamma\rangle\,(\gamma = \langle\rangle)\,\Lambda(\alpha_X :: \kappa')\lambda(x_X : \tau')\,\exists(\beta :: \kappa)M : \forall(\alpha_X :: \kappa')(\tau' \to \exists(\beta :: \kappa)\tau)\end{array}}$$

$$\textsc{ModApp}\quad \frac{\begin{array}{c}\Gamma \vdash A_1 \Longmapsto \beta_1 :: \kappa_1' \triangleright M_1 : \forall(\beta_2 :: \kappa_2)(\tau_1 \to \exists(\beta :: \kappa)\tau) \\ \Gamma \vdash A_2 \Longmapsto \beta_2 :: \kappa_2' \triangleright M_2 : \tau_2 \qquad \Gamma \vdash \exists(\beta_2 :: \kappa_2')\tau_2 \leqslant \exists(\beta_2 :: \kappa_2)\tau_1 \\ \Gamma, \beta_2 :: \kappa_2 \vdash \exists(\beta :: \kappa)\tau \equiv \exists(\beta :: \kappa')\tau' \qquad \Gamma \vdash \exists(\beta :: \kappa')\tau' :: \star \qquad \gamma, \beta_2 \sharp \operatorname{dom}\Gamma\end{array}}{\Gamma \vdash A_1\ A_2 \Longmapsto \beta :: \kappa' \triangleright \mathsf{open}\,\langle\beta\rangle\,(\nu(\beta_1 :: \kappa_1')\nu(\beta_2 :: \kappa_2')M_1\ \beta_2\ M_2) : \tau'}$$

$$\textsc{ModStruct}\quad \frac{\Gamma \vdash B \Longmapsto \left[(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i\in 1..n}\right] \triangleright [b \mid r] : [R] \qquad \tau' = \{R[\alpha_1 \leftarrow \beta.\ell_1] \cdots [\alpha_n \leftarrow \beta.\ell_n]\} \qquad b' = \nu(\alpha_1 :: \kappa_1) \ldots \nu(\alpha_n :: \kappa_n)\Sigma\,\langle\beta\rangle\,(\beta = \langle(\ell_i = \alpha_i)^{i\in 1..n}\rangle)\,(b[\{r\}] : \tau')}{\Gamma \vdash \mathsf{struct}\ B\ \mathsf{end} \Longmapsto \beta :: \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i\in 1..n}\rangle \triangleright b' : \tau'}$$

$$\textsc{ModProj}\quad \frac{\begin{array}{c}\Gamma \vdash A \Longmapsto \beta :: \kappa \triangleright M : \{(\ell_i : \tau_i)^{i\in 1..n}\} \qquad j \in 1..n \qquad \ell_j = \ell_X \\ \kappa = \langle(\ell_i \text{ as } \alpha_i :: \kappa_i)^{i\in 1..n}\rangle \qquad \Gamma \vdash \kappa_j[\alpha_1 \leftarrow \beta.\ell_1] \cdots [\alpha_{j-1} \leftarrow \ell_{j-1}] \equiv \kappa_j' \\ \Gamma, \exists\beta :: \kappa, \forall(\gamma = \beta.\ell_j :: \kappa_j') \vdash \tau_j \sim \tau_j' :: \star \qquad \Gamma \vdash \exists(\gamma :: \kappa_j')\tau_j' :: \star \qquad \beta, \gamma \sharp \operatorname{dom}\Gamma\end{array}}{\Gamma \vdash A.X \Longmapsto \gamma :: \kappa_j' \triangleright \nu(\beta :: \kappa)\Sigma\,\langle\gamma\rangle\,(\gamma = \beta.\ell_j)\,(M.\ell_j : \tau_j') : \tau_j'}$$

$$\textsc{ModSeal}\quad \frac{\Gamma \vdash A \Longmapsto \beta :: \kappa \triangleright M : \tau \qquad \Gamma \vdash S \Longmapsto \exists(\beta :: \kappa')\tau' \qquad \Gamma \vdash \exists(\beta :: \kappa)\tau \leqslant \exists(\beta :: \kappa')\tau'}{\Gamma \vdash (A : S) \Longmapsto \beta :: \kappa' \triangleright \mathsf{open}\,\langle\beta\rangle\,(\exists(\beta :: \kappa)M : \exists(\beta :: \kappa')\tau') : \tau'}$$

Figure 4.15: Translation of module expressions.

that avoids the type components of the projected module, but this can fail. As in the case of applications, if projections are restricted to *paths* in ML, then the relocation cannot fail.

Rule ModSeal To translate a signature ascription, one closes the type components of the module, then promotes it to the type required by the signature by the means of subtyping, and finally opens it again to release the modified static part to the environment. We enforce the subtyping condition through the insertion of a coercion. We could also have applied the identity function at the type described by the signature. Notice that we do not need to insert a function that deeply retypes the term, since this is implemented by the subtyping relation.

Translation of structure components (Figure 4.16 on page 127)

The judgment $\Gamma \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R]$ reads « in the environment Γ, the structure bindings B translate to the record fields r preceded by the bindings b, and the static parts have kinds that are described by the bindings ρ, and the dynamic parts have types that are described by the bindings R ». The use of the binding context b is imposed by the fact that the records of $\mathsf{F}^{\forall\omega}_{\mathsf{S}\leqslant}$ do not have internal bindings, as opposed to structures in ML. To better understand the rules, the following

invariant should be kept in mind: if $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R]$ hold and if $\exists\rho$ denotes $\exists\alpha_1 :: \kappa_1, \ldots, \exists\alpha_n :: \kappa_n$, where $\rho = \ell_1$ as $\alpha_1 :: \kappa_1, \ldots, \ell_n$ as $\alpha_n :: \kappa_n$, then $\Gamma, \exists\rho \vdash b[\{r\}] : \{R\}$ holds and the ℓ_is are pairwise distinct. A detailed description of each rule follows.

Rule StructEmpty An empty list of bindings translates to an empty list of record fields in an empty environment. Its static and dynamic parts are both empty.

Rule StructType To translate a binding type $X :: k = u$; B beginning with a transparent type declaration, one translates the rest of the binding B in an extended environment that contains the type definition, that is where the type component has a singleton kind. Since a type definition only changes the static part of the module, the resulting dynamic part is the one of the rest B, while the static part is the one of B augmented with the new type definition. As a consequence, the binding context is extended with a part that exports the corresponding static part, that is, with a Σ. Notice that in the kind of static part, the type component is nested, but not in the binding context: this is because relocation of components will happen once the structure bindings are put inside a struct ... end.

Rule StructVal To translate a binding val $X = e$; B beginning with a value declaration, one translates the rest of the binding B in an extended environment that makes the value component visible. Since a value declaration only changes the dynamic part of the module, the resulting static part is the one of the rest B, while the dynamic part is the one of B augmented with the new value definition. As a consequence, the binding context is extended with a let that defines the value component.

Rule StructMod To translate a binding module $X = A$; B beginning with a module declaration, one translates the module A and then the binding B in an environment that is extended with the static and dynamic parts of A. As module components have an impact both on static parts *and* on dynamic parts, both are extended with a nested version of the corresponding parts of the module A.

Rule StructModType To translate a binding module type $X = S$; B beginning with a module type declaration, one translates the module type S and then the binding B in an environment that is extended with the *definition* of S, that is, with a variable of a singleton kind. As for type components, the module type declaration extends the static part only.

Rule StructInclude To translate a binding include A ; B beginning with the inclusion of a module expression, one proceeds as in the case of a module declaration, except that no nesting is performed.

Translation of module types (Figure 4.17 on page 128)

Module types always translate to existential types. The judgment $\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau$ reads « in the environment Γ, the signature S translates to the type $\exists(\alpha :: \kappa)\tau$ ». The variable α of kind κ is its static part, and τ is its dynamic part. Keeping the following invariant in mind will help reading the rules: if $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau$, then $\Gamma \vdash \tau :: \kappa$ holds. A detailed description of each rule follows.

Rule SigVar To translate a signature variable, one looks up in the environment for the corresponding type variable, that should contain a definition for a signature, that is, a variable that has for kind the singleton of an existential type. The translated signature is this very definition.

Rule SigProj To translate a projection $A.X$, one checks that the projection leads to a signature, and that this signature can be used, *i.e.* does not depend on internal abstract type components. The latter condition is always met, when one restricts A to be a path. More precisely, one translates

$$\textsc{StructEmpty}\quad \frac{}{\Gamma \vdash \varepsilon \Longmapsto [\varepsilon] \triangleright [[\cdot] \mid \varepsilon] : [\varepsilon]}$$

$$\textsc{StructType}\quad \frac{\Gamma \vdash u \Longmapsto \tau :: k \qquad \Gamma, \alpha_X :: \mathcal{S}_k(\tau) \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R] \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{type}\ X :: k = u\ ;\ B \Longmapsto [\ell_X\ \mathsf{as}\ \alpha_X :: \mathcal{S}_k(\tau) \otimes \rho] \triangleright [\Sigma\ \langle \alpha_X \rangle\ (\alpha_X = \tau)\ b \mid r] : [R]}$$

$$\textsc{StructVal}\quad \frac{\Gamma \vdash e \Longmapsto M : \tau \qquad \Gamma, x_X : \tau \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R] \qquad x_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{val}\ X = e\ ;\ B \Longmapsto [\rho] \triangleright [\mathsf{let}\ x_X = M\ \mathsf{in}\ b \mid \ell_X = x_X \oslash r] : [\ell_X : \tau \oslash R]}$$

$$\textsc{StructMod}\quad \frac{\Gamma \vdash A \Longmapsto \alpha_X :: \kappa \triangleright M : \tau \qquad \Gamma, \alpha_X :: \kappa, x_X : \tau \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R] \qquad \alpha_X, x_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{module}\ X = A\ ;\ B \Longmapsto [\ell_X\ \mathsf{as}\ \alpha_X :: \kappa \otimes \rho] \triangleright [\mathsf{let}\ x_X = M\ \mathsf{in}\ b \mid \ell_X = x_X \oslash r] : [\ell_X : \tau \oslash R]}$$

$$\textsc{StructModType}\quad \frac{\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau \qquad \Gamma, \alpha_X :: \mathcal{S}(\exists(\alpha :: \kappa)\tau) \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R] \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{module\ type}\ X = S\ ;\ B \Longmapsto [\ell_X\ \mathsf{as}\ \alpha_X :: \mathcal{S}(\exists(\alpha :: \kappa)\tau) \otimes \rho] \triangleright [\Sigma\ \langle \alpha_X \rangle\ (\alpha_X = \exists(\alpha :: \kappa)\tau)\ b \mid r] : [R]}$$

$$\textsc{StructInclude}\quad \frac{\Gamma \vdash A \Longmapsto \alpha :: \langle \rho' \rangle \triangleright M : \{R'\} \qquad \rho' = (\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i \in 1..n} \qquad R' = (\ell'_j : \tau_j)^{j \in 1..m}}{\Gamma, \alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R] \qquad \alpha_1, \ldots, \alpha_n \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{include}\ A\ ;\ B \Longmapsto [\rho' \otimes \rho] \triangleright \left[\mathsf{let}\ x = M\ \mathsf{in}\ b \mid (\ell'_j = x.\ell'_j)^{j \in 1..m} \oslash r\right] : [R' \oslash R]}$$

Figure 4.16: Translation of structure bindings.

$$\textsc{SigVar}\ \frac{\alpha_X :: \mathcal{S}(\exists(\alpha :: \kappa)\tau) \in \Gamma}{\Gamma \vdash X \Longrightarrow \exists(\alpha :: \kappa)\tau} \qquad \textsc{SigProj}\ \frac{\Gamma \vdash A \Longrightarrow \alpha :: \kappa \triangleright M : \tau \qquad \Gamma \vdash \kappa \equiv \langle \ell_X \text{ as } \gamma :: \mathcal{S}(\exists(\beta :: \kappa')\tau')\ ;\ \rho\rangle}{\Gamma \vdash A.X \Longrightarrow \exists(\beta :: \kappa')\tau'}$$

$$\textsc{SigSig}\ \frac{\Gamma \vdash D \Longrightarrow \tau}{\Gamma \vdash \mathsf{sig}\ D\ \mathsf{end} \Longrightarrow \tau} \qquad \textsc{SigFunctor}\ \frac{\Gamma \vdash S \Longrightarrow \exists(\alpha_X :: \kappa)\tau \qquad \Gamma, \alpha_X :: \kappa \vdash S' \Longrightarrow \tau' \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{functor}\ (X : S)S' \Longrightarrow \exists(\beta :: \langle\rangle)\forall(\alpha_X :: \kappa)\tau \to \tau'}$$

$$\textsc{SigWith}\ \frac{\begin{array}{c}\Gamma \vdash S \Longrightarrow \exists(\alpha :: \kappa)\tau \qquad \Gamma \vdash u \Longrightarrow \tau' :: k' \\ \Gamma \vdash \mathcal{S}_{k'}(\tau') \leqslant \kappa'_X \qquad \Gamma \vdash \kappa \equiv \langle \ell_X \text{ as } \beta :: \kappa'_X\ ;\ \rho\rangle\end{array}}{\Gamma \vdash S\ \mathsf{with\ type}\ X :: k' = u \Longrightarrow \exists(\alpha :: \langle \ell_X \text{ as } \beta :: \mathcal{S}_{\kappa'_X}(\tau')\ ;\ \rho\rangle)\tau}$$

Figure 4.17: Translation of signatures.

A, and examines its static part to extract the field ℓ_X. It should be bound to the definition of a signature, that is to the singleton of an existential type.

Rule SigSig The translation of a signature is the translation of its components.

Rule SigFunctor To translate a functor type, one translates its domain, and then its codomain with the extra knowledge of the static part of the domain to produce the type of a polymorphic function. Moreover, since a functor has an empty static part, we enclose this type in an existential whose bound has the empty record kind. Functors with a non-empty static part appear when considering *applicative functors*: the static parts is then composed of type functions. Applicative functors are beyond the scope of this work.

Rule SigWith To translate a constrained module type, one translates the unconstrained module type, then tries to isolate the field to be constrained, and returns the existential type, where the bound has been constrained.

Translation of signature components (Figure 4.18 on the facing page)

The judgment $\Gamma \vdash D \Longrightarrow \exists(\alpha :: \kappa)\tau$ reads « in the environment Γ, the signature bindings D translate to the type $\exists(\alpha :: \kappa)\tau$ ». The variable α of kind κ is its static part, and τ is its dynamic part. The rules should be read with the following invariant in mind: if $\Gamma \vdash \mathsf{ok}$ and $\Gamma \vdash D \Longrightarrow \exists(\alpha :: \kappa)\tau$, then $\Gamma \vdash \tau :: \kappa$ holds. We review each rule. They are similar to the rules for the translation of structure bindings (Figure 4.16 on the previous page).

Rule SigEmpty The empty signature binding as an empty static part and an empty dynamic part.

Rule SigVal A signature binding beginning with a value specification only extends the dynamic part of the rest of the signature.

Rule SigTypeManifest A signature beginning with a transparent type definition extends the static part only, but a relocation happens in the dynamic part to cope with the nesting of the type component. Note that the static part is extended with a singleton kind to store the type definition.

Rule SigTypeAbstract This case is similar to the previous one: the only difference is that no singleton is used, since there is no definition.

$$\textsc{SigEmpty}\ \frac{}{\Gamma \vdash \varepsilon \Longmapsto \exists(\alpha :: \langle\rangle)\{\}} \qquad \textsc{SigVal}\ \frac{\Gamma \vdash u \Longmapsto \tau :: \kappa \qquad \Gamma \vdash D \Longmapsto \exists(\alpha :: \kappa)\{R\}}{\Gamma \vdash \mathsf{val}\ X : u\ ;\ D \Longmapsto \exists(\alpha :: \kappa)\{\ell_X : \tau \otimes R\}}$$

$$\textsc{SigTypeManifest}\ \frac{\Gamma \vdash u \Longmapsto \tau :: k \qquad \Gamma, \alpha_X :: \mathcal{S}_k(t) \vdash D \Longmapsto \exists(\alpha :: \langle\rho\rangle)\tau' \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{type}\ X :: k = u\ ;\ D \Longmapsto \exists(\alpha :: \langle \ell_X\ \mathsf{as}\ \alpha_X :: \mathcal{S}_k(\tau) \otimes \rho\rangle)\tau'[\alpha_X \leftarrow \alpha.\ell_X]}$$

$$\textsc{SigTypeAbstract}\ \frac{\Gamma, \alpha_X :: k \vdash D \Longmapsto \exists(\alpha :: \langle\rho\rangle)\tau' \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{type}\ X :: k\ ;\ D \Longmapsto \exists(\alpha :: \langle \ell_X\ \mathsf{as}\ \alpha_X :: k \otimes \rho\rangle)\tau'[\alpha_X \leftarrow \alpha.\ell_X]}$$

$$\textsc{SigMod}\ \frac{\Gamma \vdash S \Longmapsto \exists(\alpha_X :: \kappa)\tau \qquad \Gamma, \alpha_X :: \kappa \vdash D \Longmapsto \exists(\alpha :: \langle\rho\rangle)\{R\} \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{module}\ X : S\ ;\ D \Longmapsto \exists(\alpha :: \langle \ell_X\ \mathsf{as}\ \alpha_X :: \kappa \otimes \rho\rangle)\{\ell_X : \tau \otimes R\}[\alpha_X \leftarrow \alpha.\ell_X]}$$

$$\textsc{SigModType}\ \frac{\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau \qquad \Gamma, \alpha_X :: \mathcal{S}(\exists(\alpha :: \kappa)\tau) \vdash D \Longmapsto \exists(\alpha :: \langle\rho\rangle)\tau' \qquad \alpha_X \sharp \operatorname{dom}\Gamma}{\Gamma \vdash \mathsf{module\ type}\ X = S\ ;\ D \Longmapsto \exists(\alpha :: \langle \ell_X\ \mathsf{as}\ \alpha_X :: \mathcal{S}(\exists(\alpha :: c)) \otimes \rho\rangle)\tau'[\alpha_X \leftarrow \alpha.\ell_X]}$$

$$\textsc{SigInclude}\ \frac{\begin{array}{c}\Gamma \vdash S \Longmapsto \exists(\alpha :: \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in 1..n}\rangle)\{(\ell'_j : \tau_j)^{j\in 1..m}\} \\ \Gamma, \alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n \vdash D \Longmapsto \exists(\alpha :: \langle\rho\rangle)\{R\} \\ R' = R[\alpha_1 \leftarrow \alpha.\ell'_1] \cdots [\alpha_n \leftarrow \alpha.\ell'_n] \qquad \alpha_1, \ldots, \alpha_n \sharp \operatorname{dom}\Gamma\end{array}}{\Gamma \vdash \mathsf{include}\ S\ ;\ D \Longmapsto \exists(\alpha :: \langle(\ell_i\ \mathsf{as}\ \alpha_i :: \kappa_i)^{i\in 1..n} \otimes \rho\rangle)\{(\ell'_j : \tau_j)^{j\in 1..m} \otimes R'\}}$$

Figure 4.18: Translation of signature bindings.

Rule SigMod A signature binding that begins with a module specification extends both the static and the dynamic part. As in other cases, relocation is performed, because of nesting.

Rule SigModType This case is similar to the one of SigTypeManifest: the static part is extended with a singleton, relocation is performed on the dynamic part.

Rule SigInclude A signature binding that starts with the inclusion of a signature translates to a type, where the static parts of the signature and of the rest of the bindings are merged together, and so are their dynamic parts.

Expected properties

As for the metatheory of $\mathsf{F}^{\curlyvee\omega}_{\mathcal{S}\leqslant}$, no proof has been done about the translation. We expect the following invariants on the translation judgments:

Conjecture 4.3.1. *The image of the translation is in* $\mathsf{F}^{\curlyvee\omega}_{\mathcal{S}\leqslant}$. *That is, under the hypotheses:*

- *if* $\Gamma \vdash \mathrm{ok}$ *and* $\Gamma \vdash e \Longmapsto M : \tau$, *then* $\Gamma \vdash M : \tau$ *holds;*
- *and if* $\Gamma \vdash \mathrm{ok}$ *and* $\Gamma \vdash u \Longmapsto \tau :: \kappa$, *then* $\Gamma \vdash \tau :: \kappa$ *holds,*

then the following assertions hold:

- *if* $\Gamma \vdash \mathsf{ok}$ *and* $\Gamma \vdash A \Longmapsto \alpha :: \kappa \triangleright M : \tau$ *then* $\Gamma, \exists\alpha :: \kappa \vdash M : \tau$ *holds;*
- *if* $\Gamma \vdash \mathsf{ok}$ *and* $\Gamma \vdash B \Longmapsto [\rho] \triangleright [b \mid r] : [R]$ *and* $\exists\rho$ *denotes* $\exists\alpha_1 :: \kappa_1, \ldots, \exists\alpha_n :: \kappa_n$, *where* $\rho = \ell_1 \mathsf{\ as\ } \alpha_1 :: \kappa_1, \ldots, \ell_n \mathsf{\ as\ } \alpha_n :: \kappa_n$, *then* $\Gamma, \exists\rho \vdash b[\{r\}] : \{R\}$ *holds and the* ℓ_i*s are pairwise distinct;*
- *if* $\Gamma \vdash \mathsf{ok}$ *and* $\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau$, *then* $\Gamma \vdash \exists(\alpha :: \kappa)\tau ::$ *holds;*
- *if* $\Gamma \vdash \mathsf{ok}$ *and* $\Gamma \vdash D \Longmapsto \exists(\alpha :: \kappa)\tau$, *then* $\Gamma \vdash \exists(\alpha :: \kappa)\tau ::$ *holds.*

This conjecture gathers the invariants we gave before the explanation of each group of translation rules.

Using the translation to draw the comparison

One reason to define a translation from modules to $\mathsf{F}^{\forall\omega}_{S\leqslant}$ was get more objective arguments to compare the ML module system with $\mathsf{F}^{\forall\omega}_{S\leqslant}$. The main difference between the two is that $\mathsf{F}^{\forall\omega}_{S\leqslant}$ imposes a style where the phases are split, that is, where type components (*resp.* module types components) and value components (*resp.* module components) are not defined together, but separately, in the contrary to ML. We analyze this difference at the level of module types and then at the level of modules, with the help of our translation.

Comparison at the type-level

Module types are translated into existential types, whose bound is the static part, and whose body is the dynamic one of the signature. The difference of style is important: *a signature is dislocated into two parts*. Defining signatures in $\mathsf{F}^{\forall\omega}_{S\leqslant}$ is also more verbose when dealing with module type definitions, because they are unfolded by the translation. Notice that it would have been possible to be less verbose by exploiting type functions: for instance, instead of encoding a signature S as a type of the form $\exists(\alpha :: \kappa)\tau$, we could have chosen to use a type of the form $\lambda(\alpha :: \kappa)\,\tau$. This would have saved duplication of the dynamic part τ of the signature S, by applying it at the sites where S is used, but this would not have permitted to prevent the repetition of the static part κ. Another cause of verbosity is the lack of extensibility of record types and of record kinds, which forces to copy the included signatures (see rule SigInclude).

Comparison at the term-level

To draw the comparison at the level of terms, we analyze the main cases of the translation of modules.

Definition of functors Functors are translated into polymorphic functions that return an existential type. Parametrization of the functor in $\mathsf{F}^{\forall\omega}_{S\leqslant}$ is as verbose as in ML, but the phase-separated style splits the signature into two parts, hence the parametrization is also split into two parts: parametrizing first by the static part, then by the dynamic one. The open existential in the body has to be closed: this requires to write the kind of the static part of the body, which can be expensive. Hence, definition of functors is more verbose in $\mathsf{F}^{\forall\omega}_{S\leqslant}$ because of the existential closure of the body only.

Functor applications Since the translation turns functors into polymorphic functions, they have to be explicitly instantiated, then applied and finally opened, while just one application is necessary in ML. This is a tiny difference, thanks to the invariant we imposed, that every module has a single open existential type, *i.e.* a single access path to type components, as transparently done in ML: this invariant has for consequence that instantiations of functors are very *cheap* in terms of code size, since the static part of the module to apply, that is used to instantiate the functor, is accessible through a type variable only. Notice that subkinding plays a crucial role here: it

permits to directly give the static part as argument, whereas inserting a rekinding function would have been necessary without subkinding. Opening the result of the application is also very cheap. The main difference is the need to insert scope restrictions on the static parts of the two members of an application, to enforce our invariant on static parts: they can be costly because one has to give the kinds of the static parts that need to be restricted. We conclude that functor application is as verbose in ML as in $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$, excepted for the kinds of the needed scope restrictions.

Definition of structures The translation of a structure is verbose due to the *relocation* that is performed, because, once again, of the kinds of the restrictions, but also because of the type of the structure that has to be repeated. Due to our invariant on static parts, the creation of structures in $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ is more verbose than in ML.

Definition of structure bindings The translation of signature bindings *does not preserve the structure of programs*: one separately constructs a binding context and the record of values. This is due to the fact that the records of $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ do not have internal bindings as found in ML structures. The verbosity is also increased compared to ML for the case of module inclusions: this is due to the fact that records of $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ are not extensible (no include or with notation). Adding these features to $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$'s records would certainly allow to maintain the structure of the code and recover the same concision.

Module projections The translation of module projections involves a relocation of the static part, and is consequently verbose, since one has to give the kind of the static part of the module being projected, as well as the resulting type.

Signature ascription The translation of a signature ascription is more verbose than in ML due to the kind that must be repeated to close the static part of the module on which the signature is applied.

Almost all of the rules preserve the structure of the ML code apart from the structure bindings: by "structure" we mean the untyped skeleton, obtained by type erasure. We think this is of crucial importance, because it means that up to explicit constructs for types, one can program in $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ as in ML: your program does not have to be radically restructured, but just annotated, to fit in $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$. It also suggests that the translation we sketched just decorates your program, which is similar in effect to what type inference does. This characteristic is made possible by the use of open existential types and the use of subtyping.

The translation tells us that $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ is more verbose than ML: this is due to the explicit management of existential types, and to the fact that module type definitions are unfolded by the translation. The first cause of verbosity might be leveraged by some techniques of inference, but it is still unclear how to proceed or what would be their real benefit. The question of the unfolding of signatures may be solved by two means: first, by encoding definition of signatures to types of the form $\lambda(\alpha :: \kappa)\,\tau$ instead of $\exists(\alpha :: \kappa)\tau$, as already mentioned. This would avoid the duplications of the static parts represented by τ. A primitive notion of definitions of kinds could then be employed to share kind expressions.

$$\textsc{TermPack}\quad \frac{\Gamma \vdash A \Longmapsto \alpha :: \kappa \triangleright M : \tau \qquad \Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa')\tau' \qquad \Gamma \vdash \exists(\alpha :: \kappa)\tau \leqslant \exists(\alpha :: \kappa')\tau'}{\Gamma \vdash \mathsf{pack}\ A : S \Longmapsto (\exists(\alpha :: \kappa)M : \exists(\alpha :: \kappa')\tau') : \star}$$

$$\textsc{ModUnpack}\quad \frac{\Gamma \vdash e \Longmapsto M : \exists(\alpha :: \kappa)\tau \qquad \Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa')\tau' \qquad \Gamma \vdash \exists(\alpha :: \kappa)\tau \equiv \exists(\alpha :: \kappa')\tau' :: \star}{\Gamma \vdash \mathsf{unpack}\ e : S \Longmapsto \alpha :: \kappa \triangleright \mathsf{open}\ \langle\alpha\rangle\ M : \tau}$$

$$\textsc{TypePack}\quad \frac{\Gamma \vdash S \Longmapsto \exists(\alpha :: \kappa)\tau}{\Gamma \vdash \mathsf{pack}\ S \Longmapsto \exists(\alpha :: \kappa)\tau :: \star}$$

Figure 4.19: Translation of first-class modules.

Extending the translation to support first-class modules

In this section, we show that our translation extends to first-class modules *à la* Russo: we extend the different classes of syntax as in [RRD10].

$$\begin{array}{lcll} e & ::= & \ldots & \mid \mathsf{pack}\ A : S & \text{(Terms)} \\ A & ::= & \ldots & \mid \mathsf{unpack}\ e : S & \text{(Modules)} \\ u & ::= & \ldots & \mid \mathsf{pack}\ S & \text{(Types)} \end{array}$$

Terms of the base language are extended with the construct $\mathsf{pack}\ A : S$ that injects a module A of module type S into the class of terms. The class of modules is extended with the inverse construction $\mathsf{unpack}\ e : S$ that expects a packaged module and releases it as a module of signature S. Packed modules have a type of the form $\mathsf{pack}\ S$.

A term $\mathsf{pack}\ A : S$ actually really packs the open existential that constitutes the translation of A. Since the signature S can be greater than the signature of A, a coercion is added to promote the translation to the right type.

A module $\mathsf{unpack}\ e : S$ simply translates to the opening of the translation. The signature S is checked to be compatible with the one of the packaged module e.

A packaged type $\mathsf{pack}\ S$ is just translated into the translation of S.

Hence, extending the translation to support first-class modules is straightforward, and furthermore, the constructions are translated into similar constructs: packing corresponds to closing an existential, and unpacking corresponds to opening an existential.

Other translations to F^ω-like languages

Other translations from module calculi to languages of the same family as System F^ω have already been studied. The more recent and also the closest to our work is the "F-ing modules" translation [RRD10]. Indeed, our translation uses the same structure as the F-ing translation. There are two main differences between our translation and the one of F-ing: first, we use open existential types, whereas usual existential types are used in F-ing. The difference in the translation is that the unpack/re-pack pattern is employed, whereas we make use of the pattern that consists in closing/opening existentials when dealing with functors, but otherwise we use open witness definitions: the two patterns avoid the need to move some pieces of code. They are used at different places in the translations. Second, we take advantage of singleton kinds to prevent duplicating type definitions and also to enforce the invariant that the static part of a module can be accessed via a *single* entry point, that is an existential variable.

Shan [cS06] gives a translation of the unifying framework on modules from [DCH03] into F^ω. It supports the encoding of generative and applicative functors.

Dreyer [Dre07b] encodes a module system supporting recursive modules called RMC into his calculus RTG, that is close in spirit to $\mathsf{F}^{\forall}$ (see Section 2.7 on page 49).

Shao [Sha99] also encodes module calculi into F^ω and uses different intermediate languages to succeed.

4.4 Conclusion and future work

In this chapter, we defined the language $\mathsf{F}^{\forall\omega}_{S\leqslant}$, that is built upon the work of the two previous chapters: it merges the open existential types approach, and the singleton kinds calculus, and goes one step further, by integrating the type equivalence provided by the singleton kinds with width extensions of record types and record kinds into a powerful notion of subtyping.

No proof has currently been done on $\mathsf{F}^{\forall\omega}_{S\leqslant}$, and we do not expect them to be easy. Among the difficulties, we identified the one of extending the kind equivalence relation from dependent pairs to dependent record kinds, which does not seem to be a trivial task, although it has been rarely considered as such in the previous works on modules. Works on dependently typed records or telescopes in type theory would certainly help.

We quickly compared our language to F^ω, that it extends in three orthogonal directions, namely open existential types, singleton kinds, and subtyping. Then, we drew a more thorough comparison with an idealized ML module system by defining an encoding into $\mathsf{F}^{\forall\omega}_{S\leqslant}$. Our language brings System F^ω much closer to ML, in the sense that our encoding mostly preserves the structure of the program. To fully preserve it, it lacks better constructs to extend records and to handle dependent records. $\mathsf{F}^{\forall\omega}_{S\leqslant}$ must also be improved to reduce its verbosity: it has been acknowledged that some type information still has to be duplicated, such as in the encoding of module type definitions, or the inclusion of signatures, or the repetition of kinds for the explicit management of existential types. A system of definitions of kinds should avoid the duplication due to definitions of signatures, while adding extensible records of types to the type algebra of $\mathsf{F}^{\forall\omega}_{S\leqslant}$ should solve the problem related to the inclusion of signatures. Concerning the verbosity entailed by the explicit use of constructs for existential types, one first approach would be to infer the kinds for restrictions and closure of existentials: this would surely lighten the use of these constructs, and, at first glance, this appears to be feasible. Another, more involved approach, would be to infer most *uses* of these constructs: this is related to the open problem of type inference for existential types, which has not received much attention yet, but is known to be a difficult problem. Maybe the open existential constructs will stimulate further work in this direction.

Still, we think that $\mathsf{F}^{\forall\omega}_{S\leqslant}$ makes a great step towards a language for modular programming *à la* ML that is close, or at least based on, System F. As already discussed, it must be enhanced in some ways. A prototype implementation is in development, that will be used to experiment $\mathsf{F}^{\forall\omega}_{S\leqslant}$ on more concrete cases and, more importantly, to bring positive and negative arguments to the debate on the viability of programming in a phase-split style.

To make $\mathsf{F}^{\forall\omega}_{S\leqslant}$ resemble a full blown programming language, adding recursive types is a necessity, at least to model ML concrete datatypes. The interaction between recursive types and singleton kinds is still unknown. Recursion at the term level is also a must have, and is already discussed in Section 2.6.5 on page 42.

Chapter 5

Conclusion

The original problematics that guided our study of modules was born from the widespread remark that modules are certainly the more subtle and complex part of the ML language. The primary goal was to understand its subtleties and try to isolate the core features of modules as well as the sources of complexity.

Context

Modules are originally presented as a language where types and terms are interleaved, but that still maintains a phase separation: module types look as if they are dependent types, but it is mostly a syntactic artifact. The dependent style is preferred for convenience reasons and elegance, and most current implementations rely on this presentation.

Russo, however, showed in [Rus99, Rus03] that modules could be given non dependent types: the types of System F^ω are sufficient, or, in other words, universal and existential quantification is enough. It gave rise to the Moscow ML implementation [RRS00], that internally relies on Russo's interpretation. It has been indeed, very recently, mechanically verified [RRD10] that modules could be encoded in System F^ω. This work also shows that programming with System F^ω is, by far, less convenient than using modules. Conversely, I do not think that modules are adapted to write programs in the style of F^ω.

In any case, System F^ω gives a much simpler framework than the original module systems, and it has already been exploited in some lines of research [LCH07, Dre07a, DR08]. This is also the path we followed, and it led us to ask the question of what features System F^ω lacked, compared to ML modules. The answer consists in three items: first, the constructs for existential types in System F^ω lack flexibility; second, there is no support for type definitions in System F^ω; and in the third place, a layer of type inference is missing in F^ω. We think that these three points constitute the core features of ML modules. The latter point was not treated in this thesis, whereas the first two ones constituted the topics of Chapter 2 and of Chapter 3, respectively. Chapter 4 focused on the integration of the two preceding chapters.

Our work on modules

In Chapter 2, we proposed $\mathsf{F}^\curlyvee$ (F-zip): a language close to System F, equipped with new constructs to handle existential types. They make possible to unpack existential packages in an open scope, hence the name of *open existential types*. This essential feature permits to write programs using existential types in a style that is close to the use of *abstract types* in modules. Moreover, we showed that $\mathsf{F}^\curlyvee$ is very tightly related to F: encodings in both directions exist, that establish a static and dynamic correspondence between the two languages. We formalized in the Coq proof assistant the type soundness property of a large subset of $\mathsf{F}^\curlyvee$, and we described several extensions that would be useful to broaden the expressiveness of $\mathsf{F}^\curlyvee$ to the one of a full-blown language.

In Chapter 3, we considered the singleton kind system from [SH06]. We believe it is the tool of choice to model type definitions [Sto05], and singletons have already been used in the TILT compiler [PCHS01] and in the mechanized definition of Standard ML [LCH07], for instance. It provides a powerful notion of extensional equivalence on types. Its metatheory, however, remains involved. Stone and Harper described a normalization algorithm and a procedure to decide type equivalence, and proved them sound and complete, with the use of a logical relation. We gave another characterization, inspired by Crary's work on the elimination of singletons [Cra07]: by interpreting the unfolding of definition as an η-expansion step, we managed to define a small-step reduction relation, that mixes η-expansion with β-reduction, and that is both confluent and strongly normalizing on wellformed inputs. The key idea was to make explicit the positions where η-expansion is permitted. This way, we could define a notion of equivalence based on reduction, that is both sound and complete with respect to the original definition of equivalence of the singleton kinds system. Our completeness proof used the same technique of logical relations, thus our proof is not simpler than the one of Stone and Harper. We believe, however, that our approach is more flexible and lead to more efficient algorithms for deciding type equivalence. We hope that it can ease the understanding of the singleton kind system and, consequently, may take these techniques accessible to a wider audience.

The Chapter 4 was dedicated to the integration of the features from the two previous chapters into a single language: $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ (F-zip-full) was defined as an extension of System F^{ω} that promotes it to a level of flexibility that is very close to the one of ML modules. The main differences that remain between $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ and a module system are, first, the lack of type inference for universal and existential constructs, and, second, the fact that $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$ imposes a style where phases are split: type components and value components live in different worlds, that cannot be mixed, whereas we naturally interleave types and terms in ML structures and signatures. We think that the resulting system is a first promising step: it shows that it is possible to bring System F^{ω} very close to modules. Still, we think that we lack practical experience: the phase-split style has often been criticized, but, to our knowledge, no attempt has been made to provide tools or constructs that would help writing programs in a such phase-split manner. We do not give an answer to the practicality or difficulty of the phase-split style, but we think that, based on $\mathsf{F}^{\curlyvee\omega}_{S\leqslant}$, one could more properly compare the phase-split style with the style of modules.

Usual problems in modules

One can now more easily identify the sources of difficulty in the metatheory of modules:

Generativity of abstract types is confined to the $\mathsf{F}^{\curlyvee}$ subset: generativity corresponds to a notion of unique identifier for abstract types, and this is handled in $\mathsf{F}^{\curlyvee}$ through the use of linearity for abstract types.

Strengthening (or selfification) is restricted to the type level, in the singleton kind system, that is well understood.

Type equivalence is inherited from the type equivalence in the singleton kind system of Stone and Harper and is also well understood. In the case of the extension with records of types equipped with width subkinding (Chapter 4), because it has not been studied yet, deciding equivalence may remain a difficulty.

The avoidance problem We did not mention the avoidance problem until Chapter 4, when we added subtyping to the system. The avoidance problem is an obstacle to designing *complete* typecheckers (completeness typically relies on the existence of minimal types). This is still a difficulty for the implementer and it might cause issues for the user as well.

Generative and applicative functors constitute a pervasive notion in the literature on modules. In our study, we considered generative functors only. In Russo's interpretation, applicative functors differ from generative ones by the location of the existential quantifier in the type to specify where opening must happen, and the kind order of the quantified type variable to express dependencies. So we think that applicative functors are not an issue, and can already be handled by $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$. Extending the translation of Section 4.3.2 to support applicative functors, however, needs to be done, and belongs to future work.

Future work

Improvements, extensions and other goals have already been given in details in each chapter. We review the main objectives that we would like to reach from the work we presented.

Practice and experiments on real world examples are crucial. A prototype for $\mathsf{F}^{\Upsilon\omega}_{S\leqslant}$ is currently being developed, and we hope that the experience we would gain by using it, will confirm or invalidate our design choices. The form of type inference, especially, should be chosen and tuned with respect to these experiments.

Formalizing a greater portion of our work also belongs to the plan: it is not only a personal challenge, but it also brings benefits, when one tries to develop extensions of systems. Mechanization of properties would certainly also test and improve current techniques related to binding issues in proof assistants, for instance.

Adding dependent records to the singleton kinds system to extend its metatheory is planned, and looks pretty challenging. Relying on a machine-checked development should be a valuable help.

Handling recursive types, especially in combination with the singleton kinds system, is also considered as one of the main directions of research.

Colophon

This document was created using LaTeX 2ε using the `scrbook` class, from the KOMA-Script bundle. The TeX Gyre Pagella font was used for the text typesetting, and the Euler font for maths.

Bibliography

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, , and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15. ACM, 2008.

[AGW04] Martin Abadi, Georges Gonthier, and Benjamin Werner. Choice in dynamic linking. In *In FOSSACS'04 - Foundations of Software Science and Computation Structures 2004, Lecture Notes in Computer Science*, pages 12–26. Springer, 2004.

[Aka93] Yohji Akama. On Mints' reduction for ccc-calculus. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 1–12, London, UK, 1993. Springer-Verlag.

[Asp95] David Aspinall. Subtyping with singleton types. In *In Eighth International Workshop on Computer Science Logic*, pages 1–15. Springer-Verlag, 1995.

[AW] Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. draft.

[AZ98] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, août 1998.

[Bar84] H. P. Barendregt. *The Lambda-Calculus: its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. Elsevier, Amsterdam, The Nederlands, revised edition edition, 1984.

[Bou04] Gérard Boudol. The recursive record semantics of objects revisited. *J. Funct. Program.*, 14:263–315, May 2004.

[CC96] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the lambda-calculus with surjective pairing and terminal object. *Journal of Functional Programming*, 6(2):299–327, 1996.

[CH07] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Computation, Meaning and Logic*, 2007. Articles adedicated to Gordon Plotkin.

[CK93] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656. Springer-Verlag, July 1993.

[CL90] 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.

[Con03] Robert L. Constable. Recent results in type theory and their relationship to Automath. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, pages 1–11. Kluwer Academic Publishers, 2003.

[Coq] *Reference manual of the Coq proof assistant*, version 8.2 edition.

[Cos96] Roberto Di Cosmo. On the power of simple diagrams. In *RTA '96: Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 200–214, London, UK, 1996. Springer-Verlag.

[Cou97] 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.

[Cou98] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[Cou03] Judicaël Courant. Strong normalization with singleton types. *Electronic Notes in Theoretical Computer Science*, 70(1):53 – 71, 2003. ITRS '02, Intersection Types and Related Systems (FLoC Satellite Event).

[Cra07] Karl Crary. Sound and complete elimination of singleton kinds. *ACM Trans. Comput. Logic*, 8(2):8, 2007.

[Cra09] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *2009 Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2009.

[cS06] Chung chieh Shan. Higher-order modules in System F-omega and Haskell. Draft, May 2006.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 236–249, 2003.

[DCK93] R. Di Cosmo and D. Kesner. Simulating expansions without expansions. 0 RR-1911, INRIA, 05 1993. Projet FORMEL.

[DR08] Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 307–320, Victoria, BC, Canada, 2008. ACM.

[Dre04] Derek Dreyer. A type system for well-founded recursion. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 293–305, New York, NY, USA, 2004. ACM.

[Dre05a] Derek Dreyer. Recursive type generativity. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 41–53, 2005.

[Dre05b] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, May 2005.

[Dre07a] Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, pages 433–471, 2007.

[Dre07b] Derek Dreyer. A type system for recursive modules. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 289–302, 2007.

[Fel87] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *1998 ACM SIGPLAN International Conference on Functional Programming*, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[FG10] Alain Frisch and Jacques Garrigue. First-class modules and composable signatures in Objective Caml 3.12. Extended abstract, ML Workshop, Baltimore, Maryland, 2010.

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, 2000.

[Gog05a] Healfdene Goguen. Justifying algorithms for βη-conversion. In *FoSSaCS*, pages 410–424, 2005.

[Gog05b] Healfdene Goguen. A syntactic approach to η-equality in type theory. *SIGPLAN Not.*, 40(1):75–84, 2005.

[Gov05] Paul Govereau. Type generativity in higher-order module systems. Technical report, Harvard University, 2005.

[GP92] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing, 1992. Circulated in manuscript form. Full version in *Theoretical Computer Science*, 193(1–2):75–96, February 1998.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, New York, NY, USA, 1994. ACM.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[HP98] Martin Hofmann and Benjamin C. Pierce. Type destructors. In Didier Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[HP05] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. The MIT Press, 2005.

[HS00] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[JG95] C. Barry Jay and Neil Ghani. The virtues of η-expansion. *Journal of Functional Programming*, 5:135–154, 1995.

[KL07] Delia Kesner and Stéphane Lengrand. Resource operators for lambda-calculus. *Information and Computation*, 205(4):419–473, 2007.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. *SIGPLAN Not.*, 42(1):173–184, 2007.

[LDG+10] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.12*. INRIA, June 2010.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. *SIGPLAN Not.*, 38(9):87–98, 2003.

[Lé78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. Thèse d'état, Université Paris 7, 1978.

[Mac84] David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

[Mac86] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, New York, NY, USA, 1986. ACM.

[Mit86] John C. Mitchell. Representation independence and data abstraction. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 263–276, New York, NY, USA, 1986. ACM.

[Mit91] John C. Mitchell. On the equivalence of data representations. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pages 305–329, 1991.

[Mon10] Benoît Montagu. Experience report: Mechanizing core F-zip using the locally nameless approach. Presented at the 5th ACM SIGPLAN Workshop on Mechanizing Metatheory, Baltimore, September 2010.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[MR09] Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, January 2009.

[MR10] Benoît Montagu and Didier Rémy. Types abstraits et types existentiels ouverts. In Éric Cariou, Laurence Duchien, and Yves Ledru, editors, *Actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, pages 147–148, Université de Pau, Mars 2010.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, May 1997.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming*, pages 201–224, 2003.

[PCHS01] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon School of Computer Science, 2001.

[Pes08] Gilles Peskine. *Types abstraits dans les systèmes répartis*. PhD thesis, Université de Paris 7, Paris, France, june 2008.

[PS02] Frank Pfenning and Carsten Schuermann. *Twelf User's Guide*, 2002. Version 1.4.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 241–252, Uppsala, Sweden, September 2003.

[RRD10] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *2010 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI2010)*, January 2010.

[RRS00] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual*, June 2000.

[Rus98] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, March 1998.

[Rus99] Claudio V. Russo. Non-dependent types for standard ML modules. In *Proceedings of ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 80–97. Springer-Verlag, September 1999.

[Rus01] Claudio V. Russo. Recursive structures for Standard ML. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM Press, September 2001.

[Rus03] Claudio V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60, January 2003.

[SCPD07] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM Press, 2007.

[SH00] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 2000. ACM.

[SH06] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Logic*, 7(4):676–722, 2006.

[Sha98] Zhong Shao. Typed cross-module compilation. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 141–152, New York, NY, USA, 1998. ACM.

[Sha99] Zhong Shao. Transparent modules with fully syntatic signatures. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 220–232, New York, NY, USA, 1999. ACM.

[SML] *Standard ML of New Jersey User's Guide*.

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnivsa. Ott: Effective tool support for the working semanticist. *JFP*, 20(1):71–122, 2010.

[SP04] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–172, New York, NY, USA, 2004. ACM.

[SS] Christopher A. Stone and Andrew P. Schoonmaker. Equational theories with recursive types. Under consideration for publication in Journal Functional Programming.

[Sto00] Christopher A. Stone. *Singleton Types and Singleton Kinds*. PhD thesis, Carnegie Mellon University, 2000.

[Sto05] Christopher A. Stone. Type definitions. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 9, pages 347–385. The MIT Press, 2005.

[Tak95] Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.

[vO08] Vincent van Oostrom. Confluence by decreasing diagrams, converted. In *Proceedings of the 19th RTA (RTA 2008)*, volume 5117 of *LNCS*, pages 306–320, Hagenberg, July 2008. Springer.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 412–428, London, UK, 2000. Springer-Verlag.