Université Paris Cité

École Doctorale 386 — Sciences Mathématiques de Paris Centre

Inria

Retrofitting and strengthening the ML module system

Clément BLAUDEAU

Thèse de doctorat d'INFORMATIQUE

dirigée par Didier RÉMY

présentée et soutenue publiquement le 11 décembre 2024 devant un jury composé de :

Claudio RUSSO	Senior research scientist, DFINITY (Industry)	(rapporteur)
Jeremy YALLOP	Associate professor, University of Cambridge (UK)	(rapporteur)
Jacques GARRIGUE	Professor, Nagoya University (JP)	(examinateur)
Hugo HERBELIN	Directeur de recherche, Inria et Université Paris Cité	(examinateur)
Assia MAHBOUBI	Directrice de recherche, Inria	(examinatrice)
Didier RÉMY	Directeur de recherche, Inria	(directeur)
Gabriel RADANNE	Chargé de recherche, Inria	(invité, co-encadrant)

À Héloïse

Abstract

ML modules come as an additional layer on top of a core language to offer large-scale notions of abstraction and composition. They contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky, as it is easy to run into edge cases and irregularities. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In this thesis we propose a consolidation for the design of the type system of ML modules.

To this aim, we developed two type systems for an OCaml-like language, including both applicative and generative functors, extended with transparent ascription signatures. The first type system is called M^ω and is built on a prolific line of works that approached ML modules by translation to F^ω, the higher-order polymorphic lambda calculus. M^ω produces signatures in an OCaml-like syntax extended with F^ω quantifiers. It treats abstraction via the introduction of quantified abstract types variables and the *extrusion* of the quantifiers to make them cover the right scope. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases. The soundness of the type system is shown by elaboration in F^ω. We improve over previous encodings of sealing *within applicative functors*, by the introduction of *transparent existential types*, a weaker form of existential types that can be lifted out of universal and arrow types. This shines a new light on the form of abstraction provided by applicative functors and brings their treatment much closer to those of generative functors.

The second type system is called ZipML and is built on the *path-based*, *fully-syntactic* line of works. Unlike previous approaches, ZipML avoids the *signature avoidance problem* by introducing *floating fields*, which act as additional fields of a signature, invisible to the user but still accessible to the typechecker. In practice, they are handled as *zippers* on module signatures, and can be seen as a lightweight extension on existing signatures. Floating fields allow to delay the resolution of signature avoidance as long as possible or desired. Since they do not exist at runtime, they can be simplified along type equivalence, and dropped once they became unreachable. We give a simple criterion for the simplification of floating fields without loss of type-sharing. We present a principled strategy that implements this criterion and performs much better than OCaml. Remaining floating fields, instead of polluting the code, may disappear at signature ascription, notably when using top-level interface files. Residual unavoidable floating fields can be shown to the user as a last resort, either to be explicitly resolved by the user, or kept until link time. The correctness of the type system is proved by elaboration into M^ω, which has itself been proved sound by translation to F^ω. The language includes module type definitions that are returned in inferred types and kept as long as possible. ZipML has been designed to be a specification of an improvement over OCaml with transparent ascription, a better solution to signature avoidance, while staying close to the source language and to its implementation.

Keywords Programming languages, modularity, modules, type systems, ML, OCaml, lambda calculus, existential types, signature avoidance, signature strengthening, applicative functors, System F

Résumé

Cette thèse traite de la description formelle, de la conception et de la transformation de la couche modulaire des langages fonctionnels de la famille ML, en particulier du langage OCaml. Dans cette famille de langages, la modularité est fournie par une sur-couche spécifique séparée, qui constitue en soit un petit calcul. Bien que l'emploi courant des modules soit aisé, leur utilisation avancée révèle parfois des irrégularités et des cas limites. Par ailleurs, leur méta-théorie reste complexe, en particulier pour les preuves de sureté du typage.

Dans l'objectif de proposer une amélioration de cette couche modulaire, cette thèse présente deux systèmes de types pour un langage proche d'OCaml, contenant à la fois des foncteurs applicatifs et génératifs, ainsi que des signatures ascriptions transparentes. Le premier système de types, M^ω, s'appuie sur une ligne prolifique de travaux qui abordaient les modules ML par traduction vers F^ω, le lambda-calcul polymorphique d'ordre supérieur. M^ω produit des signatures dans une syntaxe proche de OCaml, mais qui est étendue avec des quantificateurs de F^ω. L'abstraction de types est traitée par l'introduction de variables de type quantifiées. L'extension de leur portée est simulée par un mécanisme d'*extrusion*. Une traduction inverse des signatures de M^ω vers des signatures sources est fournie, ainsi qu'une caractérisation des cas d'*évitement de signature* – un problème classique des systèmes de modules. La sureté du typage est obtenue par une élaboration bien typée des termes de modules dans F^ω. L'encodage de l'ascription dans le corps des foncteurs applicatifs est grandement simplifié par l'introduisant des *types existentiels transparents*, une forme plus faible de types existentiels qui peuvent être extraits au travers des quantifications universelles et des types de fonctions. Cela démontre que l'abstraction au sein des foncteurs applicatifs est en réalité une forme plus restreinte de types existentiels que ce qui était précédemment estimé.

Le second système de types, ZipML, est *entièrement syntaxique* et maintiens les chemins dans les signatures. ZipML utilise une approche paresseuse et préventive du renforcement de signatures. Par ailleurs, ZipML évite le *problème d'évitement de signature* en introduisant des *champs flottants*, qui agissent comme des champs supplémentaires d'une signature, invisibles pour l'utilisateur mais toujours accessibles par le vérificateur de types. En pratique, ils sont traités comme des *zippeurs* sur les signatures de modules, et constituent donc une extension relativement restreinte du langage de signatures. Les champs flottants permettent de retarder l'évitement de signature aussi longtemps que possible ou souhaité. Comme ils ne sont pas présent à l'exécution, ils peuvent être simplifiés, réordonnés ou supprimés une fois devenus inaccessibles. Cependant, la simplification des champs flottants se fait sans perte de partage. Nous présentons une stratégie qui implémente ce critère. Les champs flottants restants, au lieu de polluer le code, disparaissent lors de l'ascription, en particulier entre le fichier de code et celui d'interface. S'il reste malgré tout des champs flottants, ces derniers peuvent être présentés à l'utilisateur dans un message d'erreur. La correction du système de types est prouvée par une élaboration dans M^ω, qui a elle-même été prouvée correcte par traduction vers F^ω. Le langage inclut des définitions de types de modules qui sont retournées dans les types inférés et conservées aussi longtemps que possible. ZipML a été conçu comme une spécification d'une amélioration par rapport à OCaml avec une ascension transparente, une meilleure solution au problème d'évitement de signature, tout en restant proche du langage source et de son implémentation.

Mots clés Langage de programmation, modularité, modules, système de types, ML, OCaml, lambda calcul, types existentiels, évitement de signature, renforcement de signature, foncteurs applicatifs, Système F

Résumé substantiel

Cette thèse traite de la description formelle, de la conception et de la transformation de la couche modulaire des langages fonctionnels de la famille ML, en particulier du langage OCaml. Elle est rédigée en anglais. Cependant, le lecteur pourra trouver dans cette section un résumé substantiel en français.

Introduction

La conception, l'étude et l'exploitation de systèmes complexes, caractérisés par un nombre de composants et d'interactions tels qu'il est impossible de les appréhender tous à la fois, nécessitent des outils particuliers, que l'on regroupe sous le terme de *modularité*. Les systèmes sont ainsi découpés en modules, qui constituent des unités relativement autonomes remplissant une fonction précise, n'interagissant avec le reste du système qu'au travers d'une *interface* bien définie. Les modules peuvent être ainsi considérés comme des *boîtes noires*, dont les détails internes peuvent être ignorés pour se concentrer sur les autres éléments du système.

Dans le cadre de la conception de logiciels complexes, la tâche des architectes logiciels est précisément de prévoir une séparation en modules qui soit pertinente, notamment en essayant de maximiser la réutilisation de modules fournissant une fonctionnalité utile à différents endroits. Parmi les outils à la disposition des architectes, les langages de programmation peuvent fournir des fonctionnalités plus ou moins puissantes pour encourager la construction d'un système modulaire. Parmi la multitude de langages, une famille de langage fonctionnels se distingue par un choix technique particulier pour ses outils de modularité: les langages ML (pour «Meta-Langage »). Dans cette famille, qui contient OCaml, SML, Moscow ML, etc, la modularité est fournie par une couche séparée du reste du langage, qui constitue en soit un calcul.

Bien que l'emploi courant de la couche de modules soit aisé, son utilisation avancée révèle parfois des irrégularités et des cas limites. Par ailleurs, sa méta-théorie reste complexe, en particulier pour les preuves de sureté du typage. Dans l'objectif de proposer une amélioration de cette couche modulaire, cette thèse présente deux systèmes de types pour un langage proche d'OCaml.

Contributions

Cette thèse présente les contributions scientifiques suivantes, regroupées autour de deux systèmes de types: M^ω et ZipML.

M^ω

Le premier système de type, M^ω, est une extension de travaux existants sur la modélisation de la couche modulaire des langages ML qui s'appuient sur la traduction des signatures vers le langage F^ω, le lambda-calcul polymorphique d'ordre supérieur. Cette lignée de travaux, surnommée *F-ing* (Rossberg et al. [2014]) d'après la publication de référence éponyme, propose d'interpréter l'abstraction comme une introduction d'une variable de type abstraite quantifiée. Le quantificateur, respectivement, existentiel, universel ou lambda, indique si l'abstraction sert pour, respectivement, l'ascription, le polymorphisme, ou la définition par l'utilisateur de champs de signatures.

Ascription transparente M^ω supporte les fonctionnalités principales des modules ML, en particulier les foncteurs applicatifs et génératifs, ainsi qu'une extension de la notion d'*alias de module*: les signatures transparentes. La clarification du rôle de cette construction et de son interaction avec les foncteurs applicatifs pour le maintien de la sureté de l'abstraction est une contribution de cette thèse.

Extrusion et existentiel transparents La principale difficulté technique de l'approche *F-ing* (Rossberg et al. [2014]) est l'*extrusion* du quantificateur: après avoir été introduit au niveau du champ de type abstrait, le quantificateur doit être déplacé pour que sa portée couvre toute la zone où le type est visible. Or, si cette extrusion au travers des types d'enregistrements (qui représentent des signatures structurelles) est relativement aisée, l'extrusion au travers des types polymorphes et des types de fonctions, appelée *skolémisation*, nécessaire pour la représentation de l'ascription au sein des foncteurs applicatifs, est impossible. Une contribution de cette thèse est précisément l'introduction d'une nouvelle forme de quantification existentielle plus restreinte, dite *transparente*, pour laquelle l'extrusion au travers des types polymorphes et des types de flèches est possible. Or, M^ω démontre que cette forme restreinte est suffisante pour modéliser l'ascription au sein des foncteurs applicatifs. D'un point de vue technique, les variables existentielles transparentes maintiennent l'expression de type qui a été cachée lors de l'introduction de la variable, sans pour autant permettre de rétablir cette expression et de « rompre l'abstraction ».

Traduction inverse des signatures M^ω, comme les autres systèmes de types de l'approche *F-ing* (Rossberg et al. [2014]), produit des signatures qui contiennent des quantificateurs. La traduction inverse, vers le langage de signature d'origine où les types abstraits sont introduits par des champs de types, se heurte au problème de l'*évitement de signature*. Une contribution de cette thèse est l'énoncé de principes pour la traduction inverse, appelée *ancrage*. Ces principes donnent un cadre clairs pour les cas de succès et les cas d'échec de l'évitement de signature, en proposant une restriction *à l'ordre 1* pour les foncteurs applicatifs. Un algorithme respectant ces principes est décrit.

Signatures abstraites M^ω modélise également une fonctionnalité spécifique à OCaml et jamais traitée dans la littérature: les signatures abstraites. Ces dernières permettent l'ascription et le polymorphisme *à l'échelle* des modules. Dans cette thèse, je propose une restriction de cette fonctionnalité pour maintenir sa décidabilité: la prédicativité. Un encodage dans une extension de F^ω avec du polymorphisme de sortes est donné.

ZipML

Le second système de type, nommé ZipML, est plus unique en son genre. S'il s'appuie nécessairement sur des travaux antérieurs, il présente plusieurs mécanismes originaux qui le rendent relativement inclassable. À l'inverse de M^ω, ZipML travaille directement avec les signatures *syntaxiques*, c'est à dire celles que peut écrire l'utilisateur et qui représentent donc l'abstraction avec des champs de type abstraits. Un avantage immédiat est que l'intuition de l'utilisateur, souvent basée sur la syntaxe plutôt que sur une traduction vers un autre langage, peut être respectée. En contrepartie, le système est nettement plus intriqué que M^ω, et les preuves de ses propriétés sont plus difficiles.

Fermeture et évitement de signature Comme mentionné ci-dessus, une difficulté majeure des systèmes de modules est que la syntaxe des signatures n'est pas suffisamment expressive pour représenter le résultat de tous les calculs de modules: certains calculs d'apparence correcte n'admettent pas de signature. Pour éviter cette difficulté, M^ω utilise un langage plus

expressif, puis traite de ce problème dans un second temps, lors de l'ancrage. ZipML introduit une nouvelle construction, la *fermeture de signature*, qui permet de décrire une signature résultant d'une projection, où un certain nombre de champs sont devenus invisibles. Au lieu de disparaître et de provoquer un évitement de signature, ces derniers sont conservés comme *champs flottants* dans la fermeture de la signature. Les champs flottants sont visibles pour le vérificateur de type, mais pas pour l'utilisateur qui ne peut mentionner que les champs visibles de la signature résultante. Cette extension, qui s'inspire fortement de la technique du « zipper »en programmation fonctionnelle est relativement orthogonale au reste du langage, ce qui permet d'espérer que son ajout dans un système existant comme OCaml ne nécessite pas une restructuration complète du compilateur. Ainsi, l'introduction de fermetures de signatures pour traiter le problème d'évitement de signatures est une contribution majeure de cette thèse.

Simplification des fermetures Si les champs flottants permettent de conserver les informations de types nécessaires à la bonne formation des signatures, ils peuvent devenir inutiles au gré des opérations faites sur le module correspondant. ZipML utilise alors une opération de *simplification*, analogue à l'ancrage de M^ω, qui parcourt les champs flottants pour tenter d'effacer ces derniers au profit de champs visibles de la signature, dans les cas où cela est possible. Cette opération est un cas particulier du sous-typage, ce qui garantit sa sureté, qui par ailleurs préserve toutes les égalités de types de la partie visible de la signature. En conséquence, la simplification préserve le typage au sens suivant: simplifier n'empêchera jamais de continuer le typage. C'est une différence majeure avec la stratégie actuelle d'OCaml qui procède à une abstraction parfois excessive des champs de types concernés par l'évitement de signature.

Renforcement Les systèmes de modules syntaxiques doivent utiliser une opération de réécriture des signatures appelée *renforcement* afin d'obtenir un partage des types abstraits entre deux copies d'un même module. Cette opération est parfois présentée comme une règle de typage flottante, non dirigée par la syntaxe. Il est alors difficile de savoir exactement quelles égalités de types sont préservées ou non, ce qui peut compliquer l'implémentation ou la preuve de propriété du système. ZipML adopte une stratégie simplificatrice: le renforcement *a priori*. Ainsi toutes les signatures qui sont ajoutées dans l'environnement de typage sont renforcées immédiatement. Les types abstraits sont donc représentés par des champs pointant vers eux-mêmes. Cette stratégie évite l'ajout d'une règle de renforcement *a posteriori* et constitue une contribution de cette thèse.

Pris individuellement, le renforcement a priori pourrait paraître coûteux: il impose une réécrire à toutes les signatures, mêmes celles de modules jamais dupliqués. Or, une autre innovation technique évite ce problème: le renforcement paresseux. Plutôt que de réécrire la signature, ZipML réutilise le mécanisme de signature transparente déjà présent dans M^ω pour marquer syntaxiquement que la signature devrait être renforcée, mais en ne le faisant effectivement qu'à la demande. Le renforcement n'est donc plus couteux et peut être fait a priori sans perte de performance. La technique de renforcement paresseux a priori est une spécificité de ZipML et une contribution de cette thèse.

Conservation des définitions Les langages de modules ML offrent tous la possibilité pour l'utilisateur d'écrire des signatures, de les nommer, et d'utiliser ces noms courts plutôt que l'expression complète de la signature, qui peut être très longue. Or, dans les efforts de formalisation précédents, incluant M^ω, ces noms courts étaient effacés au profit du contenu des signatures dans le résultat du typage. À l'inverse, ZipML conserve les noms des définitions de signatures dans sa représentation interne, qui ne sont effacées qu'au besoin par une opération de normalisation. Même si ce problème, parfois dit de *pliage-dépliage* des définitions, est

connu et traité dans d'autres contextes, une contribution de cette thèse est de l'intégrer dans un système de modules.

Plan

Ce manuscrit est constitué comme suit. Le chapitre 2 sert d'introduction étendue aux systèmes de modules ML avec une attention particulière sur les mécanismes d'abstraction. Dans le chapitre 3, je présente le langage qui sert d'étude pour le reste du document. Il s'apparente à OCaml, même si quelques aspects sont traités différemment. Le chapitre 4 est consacré à la présentation de M^ω (et de F^ω). Au delà de la présentation des jugements et des règles de typage, ce chapitre contient la description de l'ancrage, ainsi que l'introduction des existentiels transparents dans la preuve de sureté du système. Il se termine par le traitement des signatures abstraites. Le chapitre 5 est lui dédié à la présentation de ZipML. Après une introduction au typage avec des fermetures, les différents jugements du systèmes sont discutés, ainsi que le mécanisme de simplification. Une preuve de sûreté par traduction vers M^ω est présentée. Le chapitre 6 contient une discussion des autres fonctionnalités des systèmes de modules, non traitées dans cette thèse, mais qui offrent ainsi au lecteur une vue d'ensemble du domaine de recherche. Enfin, la conclusion se trouve au chapitre 7.

Remerciements

Je souhaite remercier toutes les personnes qui m'ont aidé dans l'aventure scientifique – mais aussi nécessairement *personnelle* – qu'est la rédaction d'une thèse. En premier lieu, je remercie Didier Rémy, mon directeur de thèse, qui a été tout au long de ces 4 années d'une disponibilité exceptionnelle, toujours prêt à discuter de nouvelles idées ou à m'aider à venir à bout de difficultés techniques. Au delà des langages de programmation et des systèmes de types, j'ai appris auprès de lui l'exigence de la clarté dans la pensée scientifique. Je tâcherai de continuer à maintenir la même ambition de l'excellence et de la rigueur dans mes travaux futurs. Je remercie également Gabriel Radanne qui a co-encadré ma thèse depuis Lyon. Avec nos nombreuses réunion en visio et mes quelques escapades lyonnaises j'ai pu profiter de son enthousiasme à toute épreuve, de son expérience et de son approche pragmatique.

J'ai eu la chance de faire ma thèse dans l'équipe Cambium, qui m'a apporté un environnement scientifique exceptionnel, ouvert et de haut-niveau. Je remercie ses membres permanents: Damien Doligez, Yannick Forster, Xavier Leroy, Jean-Marie Madiot, Luc Maranget, François Pottier. En particulier, je remercie François qui, par sa charge de direction de l'équipe, m'a permis de réaliser ma thèse dans des conditions privilégiées jusqu'à la soutenance et qui, par la relecture de mes papiers, m'a apporté de nombreuses remarques et suggestions d'améliorations pertinentes. Merci à Florian Angeletti, que j'ai interrompu dans son travail de nombreuses fois en mentant sur le fait que ma question ne prendrait que «quelques secondes». Je remercie chaleureusement Alexandre Moine, qui a réalisé sa thèse en même temps que moi, pour nos nombreuses discussions scientifiques et son soutien dans les démarches administratives. Je remercie Basile Clément et Chiara Daini avec qui j'ai partagé mon bureau et des séances d'escalade. Je remercie également les autres membres de l'équipe, actuels et anciens: Nathanëlle Courant, Paulo Emilio de Vilhena, Rémy Seassau, Samuel Vivien, Irene Yoon.

Je souhaiterais également remercier les personnes qui m'ont aidé dans mon parcours scientifique, tout particulièrement: Natarajan Shankar, Jean-Christophe Filliâtre, Viktor Kuncak, Fenguyn Liu, Gabriel Scherer.

L'aventure de la thèse a aussi été rendue possible par le soutien familial et amical. Je remercie mes parents, Anne-Lucie et Philippe, mes frères et ma sœur, Bénédicte, Grégoire et Théodore. Je remercie également mes amis qui m'ont aidé pendant ces années de thèse, m'écoutant raconter des parties parfois cryptiques de mon sujet de recherche: Aurélien, Benjamin, Benoît, Corentin, Céline, Elliott, Étienne, Gaëlle, Louis, Pierre, Pierre, Valentin et tous les autres. Merci.

Enfin, je n'aurais pas commencé, poursuivi ou réussi à finir ma thèse sans le soutien et l'aide constante d'Héloïse, mon épouse.

Contents

1 Introduction **17**
- 1.1 Thesis overview . . . 19
- 1.2 Contributions . . . 20
- 1.3 Research Output . . . 20

2 Features and challenges of a modern module system **23**
- 2.1 Basic modularity . . . 24
- 2.2 Functors and abstraction . . . 28
  - 2.2.1 Applicative and Generative Functors . . . 28
  - 2.2.2 Abstraction Safety and Granularity of Applicativity . . . 30
  - 2.2.3 Module level equalities and aliasing . . . 31
  - 2.2.4 Aliases and transparent ascription . . . 33
- 2.3 A Key Challenge: the Signature Avoidance Problem . . . 36
  - 2.3.1 Introduction to the avoidance problem . . . 36
  - 2.3.2 Strategies . . . 38
  - 2.3.3 Avoidance with applicative functors . . . 39
  - 2.3.4 Signature avoidance in practice . . . 42
- 2.4 Module-level abstraction . . . 42
  - 2.4.1 Abstract signatures . . . 42
  - 2.4.2 Challenges of abstract signatures . . . 44
  - 2.4.3 Simple abstract signatures . . . 46

3 The ML source system **49**
- 3.1 Syntax . . . 49
  - 3.1.1 Name-spaces . . . 52
  - 3.1.2 Shadowing . . . 53
- 3.2 Semantics . . . 53

4 M^ω **55**
- 4.1 The M^ω type system . . . 56
  - 4.1.1 Overview and technical details . . . 56
  - 4.1.2 Signatures type-checking . . . 58
  - 4.1.3 Subtyping . . . 62
  - 4.1.4 Module Expressions type-checking . . . 65
- 4.2 Identity, Aliasing, and Type Abstraction . . . 67
  - 4.2.1 A source-to-source transformation . . . 67
  - 4.2.2 Derived typing system . . . 70
  - 4.2.3 Property of identity tags . . . 70
- 4.3 Rebuilding Source Signatures . . . 75
  - 4.3.1 The Expressiveness Gaps of the Source Syntax . . . 75
  - 4.3.2 The Anchoring Process . . . 79
  - 4.3.3 Properties of Anchoring . . . 85
- 4.4 The Foundations: F^ω Elaboration . . . 89
  - 4.4.1 F^ω with Kind Polymorphism . . . 89
  - 4.4.2 Encoding of Signatures . . . 90
  - 4.4.3 Sharing Existential Types by Repacking . . . 92
  - 4.4.4 Transparent Existential Types and Their Lifting Through Function Types 93
  - 4.4.5 Implementation of Transparent Existential Types in F^ω . . . 96

4.4.6 Elaboration . . . 96
4.4.7 Properties of elaboration . . . 103
4.5 Abstract signatures . . . 105
4.5.1 Key intuitions of abstract signatures . . . 105
4.5.2 Extension of F^{ω} . . . 107
4.5.3 Typing rules . . . 109
4.6 Discussion . . . 112
4.6.1 Signature artifacts . . . 112

5 ZipML **113**
5.1 Motivation and challenges of a syntactic system . . . 114
5.1.1 Signature avoidance . . . 114
5.1.2 Strengthening . . . 115
5.1.3 Lazy expansion of definitions . . . 116
5.2 An introduction to floating fields . . . 116
5.2.1 Expressivity . . . 116
5.2.2 Chaining zippers . . . 117
5.2.3 Zipper simplification . . . 118
5.3 Formal presentation . . . 120
5.3.1 Overview . . . 120
5.3.2 Grammar extensions . . . 121
5.3.3 Strengthening . . . 123
5.3.4 Path typing, resolution and normalization . . . 124
5.3.5 Subtyping . . . 126
5.3.6 Signature typing . . . 130
5.3.7 Module typing . . . 131
5.4 Resolving signature avoidance by zipper simplification . . . 133
5.4.1 Subtyping with zippers and narrow subtyping . . . 134
5.4.2 Simplification overview . . . 135
5.4.3 Dropping a field . . . 136
5.4.4 Moving away a field . . . 136
5.4.5 Splitting a field . . . 138
5.4.6 Skipping a field . . . 139
5.4.7 Simplification algorithm . . . 140
5.4.8 Preservation of typability . . . 141
5.5 Properties . . . 143
5.5.1 Soundness of ZipML by Elaboration in M^{ω} . . . 143
5.5.2 Zipper renaming, introduction, and extrusion . . . 149
5.5.3 Completeness of ZipML with respect to M^{ω} . . . 150
5.5.4 Other properties . . . 152
5.6 Discussion . . . 152

6 Advanced features **155**
6.1 Composition . . . 156
6.1.1 Hierarchical and flat composition . . . 156
6.1.2 Recursive composition . . . 159
6.2 Other features . . . 161
6.2.1 Core and module language interactions . . . 161
6.2.2 Extending the signature language . . . 163
6.2.3 Interacting with the inference of signatures . . . 165

7 Conclusion **167**

List of Figures

1 Basic modularity . . . 24
2 Applicative/Generative Functors . . . 29
3 Example of a breach of abstraction safety . . . 30
4 Example of a use-case for transparent signatures . . . 35
5 Visual representation of the avoidance problem . . . 38

6 Source language – syntax . . . 50

7 M^{ω} – Syntax . . . 56
8 M^{ω} – Wellformedness judgments . . . 58
9 M^{ω} – Signature typing . . . 59
10 M^{ω} – Example of signature elaboration . . . 61
11 M^{ω} – Subtyping . . . 62
12 M^{ω} – Module expressions typing . . . 68
13 Identity tags source-to-source transformation . . . 69
14 M^{ω} – Anchoring . . . 82
15 M^{ω} – Untagging . . . 85
16 F^{ω} – syntax . . . 90
17 F^{ω} – typing . . . 91
18 F^{ω} – transparent existentials . . . 94
19 F^{ω} – Implementation of transparent existentials as a library . . . 97
20 Coq – Implementation of transparent existentials . . . 98
21 M^{ω} elaboration - subtyping . . . 99
22 M^{ω} elaboration – typing . . . 101
23 F^{ω} extension – syntax . . . 107
24 F^{ω} extension – wellformedness . . . 108

25 ZipML – relationship between judgments . . . 120
26 ZipML – Syntax extension . . . 122
27 ZipML – Strengthening . . . 123
28 ZipML – Path typing and path resolution . . . 125
29 ZipML – Signature and type normalization . . . 126
30 ZipML – Subtyping . . . 127
31 ZipML – Signature typing . . . 130
32 ZipML – Typing rules . . . 132

33 Summary of language constructions for composition . . . 157

Chapter 1

Introduction

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise

- Edsger Dijkstra

The most complex systems that humans have built are made out of software. The exponential increase in computational power of computers have allowed software engineers to design systems of ever growing size and intricacy, fulfilling more and more tasks. Designing, operating and upgrading those systems is challenging, as our limited human brains cannot grasp them fully at once.

Complex systems are characterized by a large number of components and a large number of interactions between those components. They often display emergent (mis)behaviors [Mogul, 2006] like deadlocks, unwanted synchronization, phase change, oscillation, etc.– where no single part of the system was misbehaving. They also suffer from a propagation of effects, where local events affect the whole system.

To make complex systems manageable we break them down into smaller parts, called *modules*. Modules are self-contained units that provide a certain function and interact with the rest of the system through a *well-defined interface*. Crucially, one can *abstract* the internal details of a module and see it as black-boxes, reducing the overall complexity of the system. This makes it possible to consider the system (or parts of the system) at different *levels of abstraction*. Besides, modules can be reused throughout the system (or with other systems), which both reduces the overall complexity by factorization and reduces the cost of development and maintenance. Finally, the presence of interfaces within the system can also limit the propagation of unwanted effects (errors, malicious actors, etc.). Overall, the two core use-cases of modularity – structuring systems in functionally-relevant units and making reusable, standardized components – are both enabled by the creation of interfaces and the ability to abstract the content of a module. Finally, the actual system is obtained by combining modules together, which is called *composition*.

Examples of modular designs are everywhere. The internet relies on a network stack that is split into layers: each layers expects some properties of the previous layer and provides a specific functionality to the next. The Linux kernel is split itself into kernel modules. Programs that run over an operating system do not access the memory itself but have an *abstract* view of it. On a different domain, humans bodies, seen as biological systems, are also organized in modules. Organs are structural modules, while cells are reusable modules. Within organs, and within cells, there can be other modules. However, biological systems often have *leaky interfaces*, and effects can propagate in all sorts of ways. One could even go as far as saying that understanding the human body is hard merely because bodies are not modular enough.

Within a program, the programming language plays an important role for modularity, as it provides the tools to build and compose modules, and to write interfaces. While it is possible to write modular code and enforce interfaces only with conventions between developers, it is much more convenient, flexible, and secure to have language features to handle modules, and static guarantees that interfaces are met. In the context of software development, the need for modularity appears both for the dynamic objects, i.e., the actual modules that will make up the running program, and for developing the code itself. Typically, for the latter, developers want to be able to separate *name-spaces* to use locally-relevant short names that

could clash with unrelated synonyms. Similarly, in large programs it is essential to be able to have *separate compilation*, and not have to recompile the whole project for every change. To that aim, modules are put into *compilation units* that can be compiled separately. Therefore, those needs come in addition to the ability to create modules and interfaces and compose them.

Quickly after the first assembly languages were developed, *functions* (sometimes called *routines*) were added to factor out common code patterns, as well as the ability to break down programs in separate files. As the field of programming languages progressed, many features were implemented to provide users with modularity and abstraction. Three – non exclusive – approaches dominate in current languages: *object-oriented programming* (OOP), *type-classes* and *module systems*. In this thesis, we focus on the latter, and more specifically on the module systems of the languages of the ML family (SML, OCaml, Moscow ML, etc.), which are renowned for their expressivity and flexibility. Overall, they are niche compared with the mainstream languages like C++, Java, Js or Python. Yet, their original design – especially regarding modularity – has inspired number of other languages and researchers. Their foundational approach to structure and safety slowly but surely percolates into mainstream uses and contributes to a world with fewer bugs and better programs.

ML family of languages The ML family originated from a proof automation language for the LCF theorem prover [Milner, 1972] called the *meta-language*, hence "ML". While several dialects of ML were developed in the 1980's, an effort to write a standard for ML led to the creation of Standard ML (SML) [Milner et al., 1997]. While earlier ML dialects had limited support for modules, SML integrated the works of Harper et al. [1987] and Tofte [1990] and had a full-fledged module system, provided as a *separate language* on top of the rest of the language, called the *core-language*. The module language was corrected and improved in the revised definition of SML [Milner et al., 1990], integrating the works of Leroy [1994] and Harper and Lillibridge [1994] (among other). In parallel, the OCaml language emerged and diverged from SML, notably because the language integrated objects [Rémy and Vouillon, 1998], but also because the module language used *applicative* functors [Leroy, 1995] (instead of SML's *generative* functors) in its module language. From there, an important research effort went into extending the module systems of SML and OCaml with all sorts of features, which we discuss throughout Chapter 2.

A glimpse of ML modules In ML, the simplest form of module is a *structure*: a collection of type and value definitions that are in a shared namespace, using the keywords **`struct ... end`**. A simple structure that provides a library to handle sets of integers, called `IntSet` is given on the left-hand side below. It internally represents sets as ordered lists.

```
module IntSet = struct
  type t = int list
  let empty = []
  let rec add x s = match s with
    | [] → [x]
    | y::s' →
      if x = y then s
      else if x < y then x::s
      else y::(add x s')
  let union s1 s2 = ...
  let intersection s1 s2 = ...
end
let s0 = IntSet.add 42 IntSet.empty

(* Accepted by the compiler *)
let wrong = IntSet.union s0 [3,3,1]
```

```
module type Set = sig
  type t
  let empty : t
  let add : int → t → t
  let union : t → t → t
  let intersection : t → t → t
end

module AbsSet = (IntSet: Set)
let s1 = AbsSet.add 42 AbsSet.empty

(* Rejected by the compiler *)
let wrong = AbsSet.union s1 [3,3,1]
```

Fields of a structure can be accessed with the dot notation, as at line 13. However, there is no interface control yet: the internal details of `IntSet` are exposed. A user might wrongly consider a list of integers that contains duplicates or is not sorted as an `IntSet.t` set, as done at line 16. This would produce wrong results and might even crash. To prevent this, we need to introduce proper interfaces.

The interface of a module is called a *signature*. Users can define their own signatures with the keywords **`sig ... end`**, and give them a name with the keyword **`module type`**, as done on the right-hand side above. Abstraction of internal details can be done by forcing a module to be seen through a signature, which is called an *ascription* – it is done with the syntax `(Module:Signature)` at line 25. It can be used to hide fields (similar to the mechanism of private fields in OOP) or to make type definitions abstract. In the above example the definition of the type of sets in `AbsSet` is made abstract: it is no longer considered equal to `int list`. Abstraction forces users of the `AbsSet` module to only interact with the sets it produces through the functions provided by the module. The wrong code at line 29 would not be accepted by the compiler.

Formalization effort As the module systems of ML languages became more and more expressive, an important research effort was also put into understanding their properties. However, providing a theoretical background for the seemingly simple mechanism of abstract type fields turned out to be hard. In their foundational paper of 1985, Mitchell and Plotkin [1985] suggested to represent abstract types as *existential types*. This idea was extended by Russo [2004] who introduced the *extrusion* mechanism: existential types defined in a module are gathered and actually quantified at the top level of modules to extend their scope. Applicative functors were identified as having higher-order existential types by Biswas [1995], and the extrusion mechanism was extended to support *skolemization* [Russo, 2004; Rossberg et al., 2014]. In this setting the syntactic, user-writable signatures are elaborated in the more expressive language of types of the System F^ω– the higher-order polymorphic lambda calculus – and type-sharing between modules is expressed differently: abstract type fields actually introduce existential type *variables*, possibly higher-order, quantified in front of the signature, so that two modules that share an abstract type simply refer to the same type expression. This has culminated in the successful, so-called, *F-ing* (Rossberg et al. [2014]) line of works [Shao, 1999; Russo, 2004; Rossberg and Dreyer, 2013; Rossberg et al., 2014; Rossberg, 2018; Blaudeau et al., 2024] (among others) where most significant ML module features are specified and proven sound *by elaboration* in F^ω, thus reusing the standard meta-theoretical properties of F^ω. This is a real benefit compared with purely syntactic systems for ML modules, which use somewhat non-standard typing rules, whose meta-theory has to be redone from start, but also has proved hard to formalize, often requiring complex syntactic techniques or semantic objects.

A consolidation effort This thesis comes after many other works that aimed at formalizing and improving ML modules. We were less interested in proposing new features than in adapting existing approaches and applying them to real-world languages, with a special attention to *backwards compatibility*. Our goal was to try to find the "expressivity closure" of existing features, in order to have a language as *regular* and *predictable* as possible. We also focused on practical aspects and usability, in order to have more realistic formalization, at the price of concision.

1.1 Thesis overview

This thesis is organized as follows. In Chapter 2, we explore the main topics and features of modularity *à la* ML, with a focus on abstraction. We present informally the features and

some of the associated challenges to support them. In Chapter 3, we present the OCaml like language that we study in the rest of this thesis – called the *source* language. We give its grammar, its semantics and discuss syntactical invariants and technical details. In Chapter 4, we introduce M^ω, a type system built on the prolific line of works that approached ML modules by translation of signatures to F^ω, the higher-order polymorphic lambda calculus. We present the typing rules and its proof of soundness, along with an algorithm to translate M^ω signatures back into the source syntax (when possible). In Chapter 5, we introduce ZipML, a second type system that follows a *syntactic, path-based* approach, built around the technical innovation of *zipper signatures*. We present its typing system, proof of soundness by elaboration into M^ω, and an algorithm for zipper simplification. In Chapter 6, we discuss other features left behind in our model, to give a complete and comprehensive overview of the whole research space of ML modules beyond the treatment of abstraction. In Chapter 7, we discuss the difference between the two systems, as well as future works and perspectives.

1.2 Contributions

The work presented here led to the following main contributions:

- The definition of an OCaml-like module language with both applicative and generative functors, and a treatment of module level aliases with *transparent signatures*.
- M^ω, a type-system for this language that represents abstract types via F^ω[1] quantified variables. M^ω follows a long line of works that approached ML modules inference and soundness by elaboration in F^ω.
- The introduction of *transparent* existential types in F^ω, a weak form of existential types that can be lifted through arrow types and universal quantifiers. Transparent existentials allow for a simpler treatment of *sealing inside applicative functors* when elaborating modules into F^ω.
- An anchoring algorithm that translates M^ω-signatures back into a path-based source syntax with a principled, first-order approach to type-errors caused by *signature avoidance*.
- ZipML, another type system for the same language but that uses path-based syntactic signatures internally. ZipML is proven sound by elaboration into M^ω.
- The introduction of *zipper signatures* as a syntactic and practical way to capture an enclosing context of a signature and to delay or avoid the *signature avoidance* problem.
- A formal treatment of user-written type and module type definition that are kept during inference and in inferred signatures.
- A systematic delayed treatment of *strengthening* that relies on *transparent signatures* and that prevents useless inlining and rewriting of signatures.

1.3 Research Output

Two conference articles were integrated in this thesis:

- *Fulfilling OCaml Modules with Transparency*,
  Clément Blaudeau, Didier Rémy and Gabriel Radanne,
  Proceedings of the ACM on Programming Languages, Volume 8, Issue OOPSLA1, 2023.

[1] The higher-order polymorphic lambda calculus.

- *Avoiding signature avoidance in ML with zippers*,
  Clément Blaudeau, Didier Rémy and Gabriel Radanne,
  Proceedings of the ACM on Programming Languages, Volume 9, Issue POPL, 2025.

The OOPSLA paper presents M^{ω} and roughly corresponds to the Chapter 4. The POPL paper covers ZipML and roughly corresponds to Chapter 5. This thesis contains text that is borrowed from those articles that I co-authored with my two advisors. Apart from Section 2.1 and Section 2.3 which are partially rewritten from the introduction of those articles, the introduction, the rest of Chapter 2 and the Chapter 7 are original material. The beginning of this chapter was inspired by the content of the Principle of Computer Systems course at Epfl.

The work of this thesis also led to the discovery of actual issues in the OCaml implementation, of various severity: OCaml #10491 (incorrect abstraction of module type fields), OCaml #11441 (incorrect aliasing between to a functor parameter), OCaml #11442 (error in subtyping between invalid signatures), OCaml #11443 (silent rewrite of type information by the typechecker), OCaml #12204 (problematic interaction between strengthening and abstract module types), OCaml #13172 (cyclic definitions in recursive signatures), OCaml #13173 (generativity of module identities inside applicative functors).

Chapter 2

Features and challenges of a modern module system

In this chapter, we introduce the design space of ML-modules informally. We present the key features and the strength and weaknesses of modularity *à la ML*. While the type theory is developed more thoroughly in Chapter 4 and Chapter 5, we gathered the key design insights and present them here. This chapter serves both as a thorough introduction of the ML features that we will formally develop. Combined with Chapter 6, it gives a complete overview of the research space as well as a plan for the future of ML modules.

Features Throughout the chapter, we present both language *mechanisms*, language *properties* and language *constructs* – we refer to all three as *features*. In each section, we summarize the features using the following marks: C for language constructs, M for mechanisms that span over several constructs, and P for properties of the language. As we do both an overview of existing features and a *wish-list* of what would constitute a fully-accomplished ML-module system, we use symbols to indicate the status of each feature:

- represents largely supported and well understood features
- represents partially supported features, for which the theoretical background might be missing or the implementation incomplete
- represents new features that are not supported in current implementations, but for which we advocate

Unless stated otherwise, the status symbol applies to OCaml, but comparisons with SML, Moscow ML, or experimental languages such as 1ML are made.

Code inserts We display code inserts with the following convention:

```
This is an OCaml (or OCaml-like) code insert
This is the result of typechecking when it succeeds
This is an error that occurred during typechecking
```

Overview After presenting the basic building blocks in Section 2.1, we focus on the key mechanism of type abstraction.

In Section 2.2, we explore the interaction between abstraction and functors. It leads to the distinction between *generativity* and *applicativity*, and the *module equivalence* problem. We discuss an extension of the syntactic criterion for applicativity via the new feature of *transparent signatures*.

In Section 2.3, we focus on the interaction between abstraction and projection, which poses the (infamous) *signature avoidance problem*. We present the main approaches to this problem, as well as the solution followed by the current OCaml implementation. We discuss the need for a *type-preserving* solution, and for a mechanism to *delay* avoidance.

In Section 2.4, we move from type-level abstraction to module-level abstraction. We present the current state of the feature in OCaml, as well as a proposed restriction of *simple abstract signatures*.

The other features (composition, first-class modules, etc.) are discussed in Chapter 6.

```
1  module Complex = struct
2    type t = float * float
3    let one = (1., 0.)
4    let zero = (0., 0.)
5    let add (a,b) (c,d) =
6      (a +. c, b +. d)
7    let mul (a,b) (c,d) =
8      (a*.c -. b*.d, b*.c +. a*.d)
9    let imaginary_part (a,_) = a
10   let complex_part (_,b) = b
11 end
```

```
12 module type Ring = sig
13   type t
14   val zero : t
15   val one : t
16   val add : t → t → t
17   val mul : t → t → t
18 end
```

```
19 module Polynomials = functor (R : Ring) →
20   struct
21     type t = R.t list
22     let zero = []
23     let one = [R.one]
24     let rec add p1 p2 = match p1,p2 with
25       | x1::p1, x2::p2 →
26         (R.add x1 x2)::(add p1 p2)
27       | [], _ → p2
28       | _ , [] → p1
29     let rec mul p1 p2 = match p1 with
30       | [] → []
31       | x1::p1 →
32         add (List.map (R.mul x1) p2)
33           (mul p1 (R.zero::p2))
34   end
```

```
35 module CX    = Polynomials(Complex)
36 module CXAbs = (CX : Ring)
```

Figure 1: Basic modularity example (in OCaml). The breaks indicate that the code can be in a separate file (even though we use continuous line numbers)

2.1 Basic modularity

In this section we give an introduction to ML modules. We start with an example and then detail the main building blocks.

Example

Let us comment the example of Figure 1. We want to define a library for handling complex numbers. This library will have some internal representation of complex numbers, and provide the basic functionality (addition, multiplication, etc.). At Line 1, we define a module `Complex` by creating a new *structure* to gather the internals of the library. Structures do not have to correspond to files: we can create local modules or put several modules in the same file. Inside the structure, we define a new *type field* `type t = float * float` (Line 2) to represent complex numbers as pairs of floating point numbers. Then, we define values and functions to handle those numbers (note that OCaml uses `+.` for the addition of floats, and `*.` for the multiplication). Later in the program, users of the module can access the content of `Complex` via a *dot* projection, as in:

```
1 let c = Complex.add Complex.zero Complex.one
```

Now, we want to build another library for polynomial. In the end, we want to use it for complex numbers, but it is not specific to them. To increase code re-usability and modularity, we want to make a library for polynomials given any module that provides a minimal set of values and functions. To do that, we first define the interface of what we expect of such module. We do so at Line 12, defining the `Ring` *module type* (not module). `Ring` defines a *signature* that contains a list of type and value declarations, that places requirements on the types of fields. We leave the type binding (that decides the internal representation of the elements of the ring) *abstract* at Line 13.

We now want to use this interface for our polynomial library. We define a new module `Polynomials` at Line 19, that, unlike `Complex` is not a structure but a *functor*: a function

from modules to modules. It takes as input `R`, a module *parameter* of signature `Ring` and produces a structure. We call the signature of the parameter (here, `Ring`) the *domain* of the functor, while the produced structure is the *body* of the functor. The implementation of polynomials uses lists of coefficients for its internal representation, as defined at Line 21[1]. As we have left the type field in `Ring` abstract (Line 13), this definition is *polymorphic* and works whatever the actual implementation of `R.t`.

After this hard work, we can finally obtain our library of polynomials over complex numbers `CX` by creating a new module that is the result of a *functor application* at Line 35. But we might not be completely satisfied: the internal details of our library are exposed, i.e., any user of `CX` can see that we represent a polynomial as a `(float * float) list`. This reduces modularity and safety: later on, one might incorrectly use a list of pairs of floats as a complex polynomial, or one might write code that accidentally depend on this representation (using list operations). To prevent those, we reuse the `Ring` library, but this time to do an *ascription*: at Line 36, we create a new module `CXAbs` by copying `CX`, but we force its signature to be exactly `Ring`. Doing so, we abstract the type equality, making `CXAbs.t` no longer equal to `(float*float) list`, but *abstract*, only equal to itself. Users of `CXAbs` will only be able to handle objects of type `CXAbs.t` via the functions provided by `CXAbs`.

The constructs presented here can be combined in many ways to create more and more complex structures. Just to give an example, we can obtain polynomials in two variables by making polynomials of polynomials. We can easily create a module that calls `Polynomials` twice, by doing:

```
1 module PolynomialsXY = functor (R : Ring) → Polynomials(Polynomials(R))
```

We can even be even more abstract by going higher order, and write a functor that takes any polynomial functor and calls it twice:

```
1 module HOPolynomialsXY = functor (P : Ring → Ring) (R:Ring) → P(P(R))
```

In the rest of this section, we detail the building blocks used in this example.

Concepts

Module layer The key design choice of ML is that modularity is provided by a *separate* language layer, the *module layer*, that uses different keywords and mechanisms from the rest of the language, called the *core-language*. As a consequence, both languages can be designed with different trade-offs, adapted for programming *in the small* and *in the large*. For instance, in ML, the core-language uses a first-order type language, but is equipped with a powerful inference engine that makes it mostly implicitly typed (and yet predictable). By contrast, the module-layer has a powerful type language, but requires more explicit type annotations.

Structures The basic building block of ML modules is the *structure*, obtained with the keywords **`struct end`**. Definitions inside a structure are called *bindings*. Inside a structure, the user can bind values and define new types, each associated with a *name*. Bindings have an *open scope*: the name they define is accessible for the rest of the enclosing structure. This is especially visible for value bindings: they are made with the **`let`** keyword, but there is no corresponding **`in`** part that would limit their scope.

```
1 let x = 22        (* open scope, "x" is available *)
2 let y = 20 in x+y (* closed scope, "y" is available only in a sub-expression*)
```

[1]Type operators are written with a reverse order of applications: `int list` is a list of integers.

Type definitions Type definitions are made with the `type` keyword. OCaml offers a rich variety of type definitions: algebraic data-types (ADT), generalized algebraic data-types (GADTs) [Garrigue and Rémy, 2012], objects [Rémy and Vouillon, 1997], polymorphic variants, extensible types, parametric types, type annotations (variance, unboxing, etc.), private types, type constraints, etc. New proposals are often made to extend the type definition mechanism: the latest being (at the time of writing) modal memory management [Lorenzen et al., 2024]. The type definition mechanism sits a bit in-between the core and module layers: it is a construct of the module language but is technically supported by the core. The module language is more or less agnostic of the structure of type declarations.

Manifest types OCaml also supports *manifest types* [Leroy, 1994]: as done in the module `Complex`, one can give a name to any *already-existing* type expression (here, `float*float`). In practice, this feature is essential, as it allows to use locally meaningful names for type expressions. It also acts as a light form of abstraction.

Signatures The type of a module – obtained by inference or written by the user – is called a *signature*. The signatures of structures are called *structural signatures* and use the delimiters `sig end`. Definitions inside a structural signature are called *declarations*[2]. Signatures can be named via a *module-type* binding. This is especially useful, as signature can grow very large and clutter the output when no names are used. While the nomenclature in the literature can vary, in this thesis, we refer to the type of modules as signatures, while module-types only refer to the binding/declaration.

Type Abstraction The killer feature of ML modules, that is also the crux from a formalization point of view is *type abstraction*. Type declarations can be left abstract. This allows the definition of *new types* that are only compatible with themselves. Abstract types provide a powerful and flexible mechanism for *encapsulation*, that is the heart of the discussion throughout the rest of this chapter.

Controlling the outside view of a module Signatures can be used to control interactions between modules in two ways. First, the *outside view* of a module can be restricted to protect internal invariants by an explicit *ascription* to a given module type. It requires a subtyping check between the inferred signature of the module expression (here, the inferred signature of `Complex`) and the user-provided one (here `Ring`). This check is *structural*: there is no need to mention the `Ring` signature at the definition point of `Complex`, it is sufficient to have the appropriate fields. By contrast, with nominal subtyping, the subtyping relations must be made explicit up front, which is less flexible and imposes an order on the definitions of module and interfaces.

Ascriptions can be used to (1) *hide* fields (making them inaccessible from the outside), (2) *reorder* fields (regardless of the original order of definitions), and (3) *abstract* type components, which hides the underlying implementation while keeping the name visible. The basic form of ascription is also called *opaque ascription*—we will later encounter *transparent ascription* in Section 2.2.4. Here, `CXAbs.t` is an available type, but its implementation as a list of pairs of floats is hidden.

Controlling dependencies of a module Signatures can also be used to restrict how a given module depends on (i.e., uses) other modules. This is achieved by turning the module into a *functor*. Here, `Polynomials` is a functor that can take any implementation `R` satisfying

[2]The terminology is not fully stabilized in the literature regarding the distinction between *bindings* for fields of structures and *declarations* for fields of structural signatures. In some works like [Rossberg and Dreyer, 2013], bindings are called *definitions*, and declarations are called *specifications*.

the `Ring` interface and that returns an implementation of the ring of polynomials over `R`. The *body* of the functor is *polymorphic* with respect to the abstract type fields of its argument, and thus, does not depend on their actual implementations. Functors can then be called and composed: `Polynomials` can be applied to modules satisfying `Ring` such as `Complex` but also the output of `Polynomials` itself. There is a subtyping check at functor application, which induces a run-time coercion.

Hierarchy Modules can be nested inside other modules as *sub-modules*. Correspondingly, submodules can be extracted by *projection*: `M.X` extracts the submodule `X` from `M`.

Higher order functors As expected by users of functional languages, functors can be *higher order*: they can take other functors as arguments. Historically, higher order-functors have been a technical challenge, especially before the introduction of applicative functors (see [Shan, 2004; Dreyer et al., 2003; Leroy, 1995; Biswas, 1995; Harper and Lillibridge, 1994; Harper et al., 1989], among others).

Overall, we have the following basic features

- ✅ M **Basic modularity**: structures, signatures, value bindings/declarations, submodules, type and module-type bindings/declarations.
- ✅ C **Type manifests**: creating new names for already-existing types
- ✅ M **Abstract types**
- ✅ M **Interface control**: ascription (interface of exports) and functors (interface of imports)
- ✅ M **Higher-order functors**

Regularity and robustness A good property for languages, especially in the context of modularity, is its *regularity*: the user should be able to arbitrarily nest and use the features, without restrictions such as being *at top-level*. It is partially the case in OCaml: submodules can be arbitrarily nested, ascription can be used at any point, functors applications can be written with an anonymous functor and anonymous argument, functors can be higher-order. However, other constructs, such as projections, are syntactically restricted to module identifiers (we expand on the technical reasons behind this choice in Section 2.3). This is detrimental for the usability of the system, as users might run into hard-to-predict unsupported edge-cases in more advanced applications.

Another important property is *robustness*: typechecking should be predictable and semantically insignificant changes should not break it. OCaml modules suffer from a lack of robustness when using the advanced features of the language. As an example, they support neither *let-binding* nor *inlining of definitions*! Let-binding can break the syntactic criterion of applicativity, which we discuss in Section 2.2.3—it can be fixed with *transparent signatures*. Inlining the definition of a submodule (or *let-inlining*) can trigger signature avoidance, as discussed in Section 2.3. This breaks the *subject reduction* property of the intuitive call-by-value semantics. While we do not necessarily expect the language to have full-blown *subject reduction*, the semantic reduction that can be expressed in the language should not break type-checking.

- 🟠 P **Robustness** the typechecking should support semantically insignificant changes.

Summary At a high-level, ML modules can be understood as a small separate calculus built on top of a source language, with standard features re-branded with ML nomenclature. If we forget about typing, *structures* are basically records: *bindings* are record-field expressions and

declarations are record-field types. *Projection* is just record access. *Ascriptions* are explicit type coercions, and *functors* are just functions. However, in addition, signatures contain abstract types. The crux of ML modules is the interaction between abstraction and other features, which we discuss in the following sections.

Other introductions to ML-modularity can be found in Dreyer et al. [2005]; Russo [2004]; Leroy [2000] (among others) or in [Pierce, 2004, §8].

2.2 Functors and abstraction

Functors are the key tool to parameterize a module with another module in a polymorphic way. The module parameter can be chosen at another point than the definition of the functor, at the *functor call* site. However, this seemingly simple feature interacts with abstraction in subtle ways.

2.2.1 Applicative and Generative Functors

Both modules and functors can be used to either *structure* the code base or to build *reusable components*. In the latter case, several instances of a functor application may be available in the context when combining different pieces of code. This is typically the case for modules providing common data-structures such as lists, hash-tables, sets, etc. When such functor produces abstract types, a question arises: should every instance of the same application produce compatible abstract types, i.e., types considered equal by the typechecker? This question leads to the distinction between *applicative* and *generative* functors, which have different semantics and correspond to different use cases. Both are supported by OCaml and illustrated in Figure 2. If two instances have *equal* abstract types, they are effectively *compatible* and the functions and values from each module can be used together. We say that two instances are *incompatible* when they have different abstract types.

Generative functors Applying a *generative functor* twice *generates* two incompatible modules, with incompatible abstract types. The body of such functor might be stateful, emits effects, or dynamically choose the implementations of its abstract types (using *first-class modules*). Generativity can also be used by programmers as a strong abstraction barrier to force incompatibility between otherwise pure and compatible data-structures that represent different objects in the program. OCaml *syntactically* distinguishes generative functors from applicative ones by requiring the last argument to be a special unit argument "`()`". Therefore, generative and applicative functors are *syntactically* distinguished: the unit argument () is not the same as an empty structure. We find this effect-suggesting syntax of taking a unit argument fitting, as calling a generative functor can indeed produce effects.

Applicative functors Conversely, applying an *applicative functor* twice with *the same argument* produces compatible modules, with the same abstract types. The body of such a functor must be pure and have a static implementation of its abstract types. In OCaml, it is left to the user's responsibility to mark impure functors as generative. Indeed, the typechecker does not track effects, but only prevents unpacking of first-class modules and calling generative functors inside the body of applicative ones. Applicativity acts as a weaker abstraction barrier, making several instances of the same structure compatible. This is especially useful to provide generic functionalities (such as hash-maps[3], sets, lists, etc.) that may appear in several places and yet be compatible. Applicative functors are the default in OCaml.

[3] `Hashtbl.Make` is actually pure, as it does not produce a new hash table itself, even though it contains impure functions.

```
module Tokens () = (struct
    type t = int
    let x = ref 0
    let fresh () = let n = !x in x := n + 1 ; n
    let eq x y = (x = y)
  end : sig
    type t
    val fresh : unit → t
    val eq : t → t → bool
  end)
module PublicTokens = Tokens()
module PrivateTokens = Tokens()
(** PublicTokens.t =/= PrivateTokens.t *)
```

(a) A generative functor — OCaml functors are made generative by having `()` as their last parameter. Here, each application of the `Tokens` functor produces a module with its own internal state that generates fresh tokens independently.

```
module OrderedSet (E:Ordered) = (struct
    type t = E.t list
    let empty : t = []
    let rec add x s = match s with
    | [] → [x]
    | y::s → if (E.lt x y) then (x::y::s)
                  else (y::(add y s))
  end : sig
    type t
    val empty : t
    val add : E.t → t → t
  end)
module S1 = OrderedSet(Integers)
module S2 = OrderedSet(Integers)
(** S1.t === S2.t *)
```

(b) An applicative functor — Functors are applicative by default in OCaml. Here, `OrderedSets(E)` is a module implementing (ordered) sets of elements of type `E.t`. Applicative functors can be used in paths directly, leading to `S1.t = OrderedSets(Integer).t`.

Figure 2: Examples of generative and applicative functors.

```
module S1 = Set.Make(struct type t = int let lt = (>) end)
module S2 = Set.Make(struct type t = int let lt = (<) end)

let s = S1.(empty |> add 1 |> add 3 |> add 2) (* [1,2,3] *)
let s' = S2.add 2 s1 (* [2,1,2,3] *)
let test = S1.mem 2 s' (* false *)
```

Figure 3: A simple example of a breach of abstraction safety. With the static equivalence criterion, this code type-checks but misbehaves. Specifically, the implementation of sets as lists assumes that the list is ordered with respect to the comparison function. However, `S1` and `S2` do not have the same comparison function, but still have compatible abstract types.

One could think that applicative functors are not *necessary*: the programmer could always have factorized in advance the functor applications that happen to be duplicated and only use generative functors. Yet, applicative functors allows for clean and concise patterns which prevents program-wide changes (even deep inside libraries). Since they where introduced by Leroy [1995], they became an appreciated feature of OCaml, contributing to the success of the language. Generative functors, the default in SML, were added to OCaml only later on (OCaml 4.02).

- ✔ C **Generative functors**: necessary for effectful modules or dynamic choice of implementation (using first-class modules, see Section 6.2.1)
- ✔ C **Applicative functors**: crucial for sharing of pure functors without *a priori* factorization of similar applications
- ✚ M **Effect tracking at the core level**: applicative functors bodies should be pure. As of now, it is left to the user responsibility, whereas a type-system check of purity would provide a much stronger guarantee.

2.2.2 Abstraction Safety and Granularity of Applicativity

A key design point is the *granularity* of applicative functors: under what criterion should two applications of a functor produce compatible modules? This problem was coined the *module equivalence problem* by Dreyer [2007a]. We say that two modules are *equivalent* when applying the same functor to both yields compatible abstract types.

An option, used in Moscow ML, is to consider modules to be equivalent when they have the same type fields (same names and same definitions[4]). This criterion is called *static equivalence* [Shao, 1999; Russo, 2000; Dreyer et al., 2003]. It is *type-safe*: in the absence of first-class modules, which are forbidden in applicative functors for that very reason, the actual implementation of the abstract types produced by the functor can only depend statically on its parameters, thus only on its *statically known* type fields. Therefore, requiring static equivalence is sufficient to ensure that the abstract types produced by the functor are the same.

However, the static equivalence criterion can make two functor applications compatible while they actually have different internal invariants. In the example of Figure 3, the lists used to represent sets are ordered with respect to the comparison function of the argument of the functor. Therefore, a typechecker implementing the static equivalence criterion would allow a user to mix sets ordered with different ordering functions, which would produce wrong results – but not crash.

[4]This relies on some equality of types. For instance, it could be the cloture of syntactic equality and applicativity – which might be problematic for recursive signatures with applicative functors.

Yet, developers often expect a stronger property called *abstraction safety*: abstract types should protect arbitrary local invariants that may also depend on values. In the example of Figure 3, applications of `OrderedSets` should produce compatible abstract types only when both the type `E.t` and the values, in particular the comparison function `E.lt`, are the same. Crucially, abstraction can protect invariants that cannot be expressed in the type-language. This is actually a key point when comparing the module system of OCaml and that of languages with very expressive type-systems, especially Gallina (the language of Coq) which has an ML like module system. In the latter, all the properties of objects can be expressed directly as lemmas (or using dependent types) and preserved by the type-system. Abstraction only serves to limit the reach of internal details.

To preserve abstraction safety, we need *dynamic equivalence*, i.e., the runtime equality of values. Unfortunately, it is undecidable in general. Besides, tracking even an approximation of the equality of value fields would be too fine-grained and cumbersome, as modules may have numerous value fields. To enforce abstraction safety while remaining practical, OCaml follows a *coarse-grain approach*: tracking equalities only at the module level (not at the value level). This was originally introduced as a *syntactic criterion* by Leroy [1995]: two functor applications produce the same abstract types when they are syntactically identical. The syntactic criterion therefore acts as a gross approximation of dynamic equivalence, preserving abstraction safety.

> ✅ P **Abstraction safety**: type abstraction should protect arbitrary invariants. While easy to verify for the rest of the language, applicative functors need a special treatment: their module equivalence criterion should be a subset of dynamic equivalence.

As a summary, applicativity of functors relies on a criterion for *module equivalence*. We distinguish two notions of equivalence:

- *Static equivalence*: the two modules have the same type fields (for some notion of type equality). Using static equivalence for applicativity yields *type-level applicativity*.
- *Dynamic equivalence*: the two modules have the values at runtime. A subset of dynamic equivalence can be tracked statically by the type-system, either (1) by tracking equality between value fields, yielding *value-level applicativity*, or (2) by tracking equality between modules, yielding *module-level applicativity*.

2.2.3 Module level equalities and aliasing

The *syntactic criterion* is however somewhat fragile, as, for example, it does not support naming sub-expressions. In this subsection, we present the OCaml extension of *module aliases* that extends this criterion in a somewhat restricted way.

Two paths with functor applications are considered equal when they are syntactically equal:

```
module F (_:sig end) = struct end
module G (_:sig end) = struct type t end
module X = struct end
let f : G(F(X)).t → G(F(X)).t = fun x → x (* typechecks *)
```

Unfortunately, naming a sub-expression breaks the syntactic equality:

```
module FX = F(X)
```

```
let f': G(FX).t → G(F(X)).t = fun x → x
```

```
Error: This expression has type G(FX).t but an expression was expected of type
         G(F(X)).t
```

Here, the type of module `FX` should manifest its module-level equality with `F(X)`. Interestingly, the OCaml feature of *module aliases* partially covers this need, even though it was historically introduced for unrelated reasons.

Module aliases To allow for a better management of name-spaces, *type-level module aliases* were added to OCaml: the signature language has been extended with the *alias declaration* construct `module` $X = P$ to express that a module is a *statically known alias* of the module at the path P. The main motivation behind this extension was to provide a mechanism to give short names to modules without actually duplicating the module at runtime. It is especially useful for accessing the standard library:

```
module M = struct
  module A = StdLib.Array (* example of short name with module aliases *)
  (* ... *)
end
```

Internally, the compiler maintains a flag to decide if the submodule can be removed from the structure at runtime (*absent alias*, also called *static alias*) or should be kept (*present alias*, also called *dynamic alias*). As it was designed as a namespace feature, it made sense to try to prevent having an actual field for the aliased module if not necessary. However, it effectively merged in the same syntax two very different features, and the user cannot easily control if the aliased module is present or absent at runtime.

This opened the door to module-level equalities, similar to the abandoned structure sharing mechanism of SML'90. In particular, the syntactic criterion was extended to include module aliases (both static and dynamic), making the following code typecheck:

```
module F (_:sig end) = struct type t end
module X = struct end
module X' = X
let f : F(X).t → F(X').t = fun x → x
```

Restrictions However, the motivating example at the beginning of this section still does not typecheck in OCaml: the aliasing feature has been designed for static aliases and is therefore quite limited. In practice, for a declaration `module M = P`, the path `P` has the following restrictions (extract of the OCaml documentation [Leroy et al., 2024])[5]:

1. it should be of the form `M0.M1...Mn` (i.e. without functor applications);
2. inside the body of a functor, `M0` should not be one of the functor parameters;
3. inside a recursive module definition, `M0` should not be one of the recursively defined modules.

The restrictions 1 and 3 were put in place to ensure that the aliased module is already dynamically present, to make the static redirection meaningful. But these two restrictions are not necessary for dynamic aliases! (Restriction 2 is discussed in the next subsection)

Overall, we advocate for a separation of the two mechanisms for aliasing. The simpler static aliasing is well suited to remain a *declaration* made inside an enclosing module, and the aliased module cannot contain functor application, other recursive modules, or functor parameters: it should have a different syntax than dynamic aliases (discussed in the next subsection).

> **C Static aliases**: to manage name-spaces (provide short names for modules), the user should be able to define *static aliases* to other, already defined, modules. Static aliases do not induce a copy of the aliased module, they are removed at runtime.

In the rest of this thesis, "aliases" refer to dynamic aliases by default.

[5] https://ocaml.org/manual/5.2/modulealias.html

2.2.4 Aliases and transparent ascription

In this subsection, we present a technical issue with aliases, solved by the introduction of a new feature of *transparent signatures* that provides a robust extension of the syntactic criterion of applicativity.

Unlike static aliases, dynamic aliases don't have to be restricted to paths without applications and could contain other recursive modules. Besides, the aliasing does not have to be at the declaration level, but could very well be at the signature level. However, paths containing functor parameters are an issue[6]. As an example, let us consider the following code:

```
1 module X : T = (* ... *)
2 module F (Y:S) = Y (* reexport *)
3 module X' = F(X)
```

The module `X'` **cannot** be given the expected alias signature:

```
1 module X' : (= X)
```

which contradicts the substitution-based intuition. Indeed, if the type system were to maintain module aliases through functor calls, it would impose strong constraints on the compilation of structures and functors that would drastically affect the performance trade-offs of modules.

To see why, let us consider the compilation of the body of the functor `F`. To enable separate compilation, it should not depend on the rest of the code, but only on the parameter signature `S` and the code of the body. In order for the body of the functor to be an alias of the argument, it should contain *all* of the fields of its parameter. However, as we allow implicit ascription at functor call, the actual argument (here `X`) might have strictly *more* fields than the ones indicated in the signature `S`, and in an *order that might be different*. Therefore, the static view of `Y` is a (potentially) strict supertype of the dynamic structure given at functor application. Yet, for the compilation of the body, we rely on the static view to compile accesses inside `Y`. The possible mismatch between the static and dynamic view can be resolved in two ways:

- We can force the static and dynamic view to match by removing the implicit ascription at functor call. In our example, it would require to rewrite the line 3 as:

  ```
  1 module X0 = (X:S)
  2 module X' = F(X0)
  ```

  ```
  1 module X0 : S
  2 module X' : (= X0)
  ```

  Besides the fact that would greatly limit the usability of functors, it would still not give the right answer: `X'` would be an alias of `X0`, not `X`.

- Or, we can choose a compilation scheme where the mismatch is not an issue: where the actual dynamic object can have any subtype of the static view. That is, a compilation scheme with *code-free* subtyping. Typically, using dictionaries with a *dynamic dispatch* through an access-table provides a code-free subtyping representation of structures, at the cost of slower accesses (due to the indirection). However, in all ML-module systems, including SML and OCaml, structures are compiled via *static dispatch*: accesses inside a structure are made with fixed offsets to be fast. In such setting, subtyping is not code free, and explicit coercions are inserted at functor calls, which break module aliases.

Therefore, the type system does not follow the naive substitution semantics for aliases inside functors, but instead uses a set of syntactically-based restrictions to prevent them in functor

[6] This issue stalled the pull request OCaml #10435 that aimed at extending the alias mechanism

signatures. Those restrictions are not stable under substitution and can be bypassed in some edge-cases[7].

> **M Static dispatch**: "*modules are often accessed but rarely reordered*", the type system should be compatible with a static dispatch compilation scheme that favor fast access over code-free subtyping.

Transparent ascription to the rescue Interestingly, *transparent ascription*, originally introduced as a module expression written `(M : S)` in SML, helps lifting this restriction. It restricts the outside view of a module `M` to the fields present in the signature `S` while preserving all type equalities. However, this feature does not increase expressiveness in SML as a similar result could be obtained via a *usual* (opaque) ascription with a signature where all type equalities have been made explicit. In OCaml however, it would increase the sharing with applicative functors. A proposal for OCaml[8] is to add transparent ascription as an extension not only of the module language, but also of the *signature* language, writing `(= P < S)` for the signature of a module that is an alias of `P` but restricted to the fields of `S`, which we call a *transparent signature*. A module with such a signature has a *module-level equality* with `P` and the content `S`. We say that it shares the same *identity* as `P`.

Transparent signatures provide a generalization of aliasing, storing both the aliasing information *and* the actual signature (hence, the memory representation). The transparent ascription *expression à la* SML `(P : S)`, is then just syntactic sugar for an opaque ascription with a transparent signature `(P :> (= P < S))` (in SML syntax).

Thanks to transparent signatures, aliasing information can be preserved through the implicit ascription at functor calls. As OCaml features applicative functors (unlike SML), this would increase the expressiveness of the signature language. Besides, concrete signatures are compatible with static dispatch and copying at function calls allows deletion and reordering of fields while preserving type equalities. The motivating example of the beginning of this section would give:

```
module X : T = (* ... *)
module F (Y:S) = Y (* re-export *)
module X' = F(X)
```

```
module X : T
module F : (Y:S) → (=Y<S)
module X' : (= X < S)
```

Another, more realistic example of code pattern where transparent signatures are necessary is given in Figure 4. This pattern arises when combining a functor (`Make3D`) that re-exports its argument (`K`) with an applicative functor called twice (`Set`), once on the argument and once on the re-exported argument `V.Scalar` so that the modules resulting from both applications may interact. Here, the fixed interface `VectorSpace` could not be *functorized* to make explicit the dependency over the underlying field, as not all vector spaces are functors over a field. Besides, type-level sharing is not sufficient to obtain the right type equalities when calling the `Set` functor.

> **C Transparent signatures (dynamic aliases)**: to extend the syntactic criterion of applicativity in a manner that composes well with functors, we propose to store both the module level equality and the dynamic signature using transparent signatures.

[7] See the following issues: OCaml #7818, OCaml #2051, OCaml #10435, OCaml #10612 and OCaml #11441.

[8] OCaml #10612. Transparent ascription is written `(P :> S)` in OCaml #10612, the opposite of the SML convention.

```
1  (* Interface definitions *)
2  module type Field = sig ... end
3
4  module type VectorSpace = sig
5    module Scalar : Field
6    ... (** more fields *)
7  end
8
9  (* Basic data-structure *)
10 module Set(Y: sig
11   type t
12   val compare : t → t → int
13   end) = struct ... end
14
15 (* Extension of a vector space
16    with additional features *)
17 module LinAlgebra(V:VectorSpace) =
18 struct
19   ...
20   module SSet = Set(V.Scalar)
21   ...
22 end
```

```
23 (* Special case of vector space
24    built directly from a field *)
25 module Make3D(K:Field) = (struct
26   module Scalar = K
27   ... (** built from K *)
28 end : sig
29   module Scalar : (= K < Field)  ...
30 end)
```

```
31 (* Implementation of real numbers ℝ *)
32 module Reals = ...
33
34 (* Vector space ℝ³ with extensions *)
35 module Space3D =
36   LinAlgebra(Make3D(Reals))
37
38 (* Are sets of reals the same
39    as sets of scalars of ℝ³ ? *)
40 let id : Space3D.SSet.t → Set(Reals).t
41    = fun x → x
```

Figure 4: An example of code pattern where transparent signatures are necessary. On the left-hand side, `VectorSpace` defines an interface for vector spaces which contains a sub-module `Scalar` for the field of scalar numbers. The functor `LinAlgebra` (line 18) uses a vector space to define linear algebra operations, one of them using sets of scalar numbers. At some other point in the development (line 25), 3D vector spaces are built directly from any field `K` via the functor `Make3D`. Its signature contains a transparent signature on its parameter `K`. Finally, on line 36, the module `Space3D` implements linear algebra for the vector space $\mathbb{R}^3$. We want the inner sets `Space3D.SSet.t`, and `Set(Reals).t` to be compatible. This requires the aliasing information to be kept between the parameter and the body of the functor `Make3D`. Currently, OCaml fails to share and identify the types `Space3D.SSet.t` and `Set(Reals).t`.

This would lead to a much more robust notion of module equivalence that extends the syntactic criterion.

- **Robust let-binding with submodules**: using transparent signatures, the user should always be able to bind module expressions as submodules without breaking type-checking.

Transparent signatures can be seen as a special case of the more general module sharing mechanism of *F-ing* (Rossberg et al. [2014]) `like P`: the transparent signature `(= P < S)` could be obtained as `(like (P : S))` (with a transparent ascription of `P` by `S`). This sharing mechanism relies on a general notion of *semantic* paths that stand for any module expression that does not introduce new abstract types whereas OCaml uses more restrictive *syntactic* paths. Therefore, module sharing signatures with semantic paths could also play the role of transparent signatures. However, we argue that semantic paths are *too powerful for their own good*. By injecting arbitrary module expressions in signatures, it makes it hard for the user to know if two modules are equivalent.

As transparent signatures enable more sharing of identities in signatures of inferred modules, aliasing becomes a good static approximation of that equivalence.

2.3 A Key Challenge: the Signature Avoidance Problem

In this section, we present the *signature avoidance* problem, a key issue of ML module systems that comes from the interaction between abstraction and projection. In Section 2.3.1, we introduce the signature avoidance problem through examples. In Section 2.3.2, we discuss possible strategies to handle it. We give some considerations for the design of an algorithm that can be used to *solve* the avoidance problem, which we call *anchoring*. In Section 2.3.3, we propose a restriction of anchoring in the presence of applicative functors for improved usability. We conclude with some practical aspects in Section 2.3.4.

2.3.1 Introduction to the avoidance problem

At its core, signature avoidance originates from the interaction of three mechanisms. First, type abstraction creates new types that are only compatible with themselves (and their aliases). Then, sharing abstract types between modules, which is essential for module interactions, produces inter-module dependencies in signatures. Finally, projection (and other mechanisms) allows the user to hide a type or module components, which can *break* such dependencies: types can be removed from the scope while they are still being referenced. Sometimes, no possible signature exists for a module in the source syntax; other times there are several incomparable ones (no principal source signature). A detailed overview of the avoidance problem can be found in [Crary, 2020].

In the literature, the nomenclature is not fully stabilized: sometimes the *avoidance problem* refers to cases where no source signature exists, sometimes to the process of finding a signature that avoids a certain type. In this thesis, we use *signature avoidance* or *the signature avoidance problem* to refer to the challenge of finding a signature that avoids a certain type, if it exists. The algorithm that actually finds such signature is called the *anchoring algorithm*.

We show here all language constructs that can hide components from the context and therefore require to avoid them. The last two use features that are discussed later in the thesis and can be skipped; we present them for completeness.

Anonymous projection The most general form is the anonymous projection (which is not valid OCaml syntax):

```
module M = (struct
  type t = A of int | B
```

```
  module X = struct
    let x = A(42)
  end
end).X
```

The module `M` is built by projecting *only* the submodule `X`, which exposes dangling dependencies on a type `t` that has become inexpressible. There is no way of writing a signature for `M` that *avoids* `t` (unless `x` is also dropped):

```
sig x : ? end
```

OCaml disallows *anonymous projections* precisely to prevent this type of patterns.

Anonymous functor call However, OCaml does allows anonymous functor call, which leads to similar situations:

```
module M =
  (functor (Y:sig type t end) → struct
    let x : Y.t list = []
  end)(struct type t = A | B end)
```

```
Error: This functor has type functor (Y : T) → sig val x : Y.t list end
       The parameter cannot be eliminated in the result type.
       Please bind the argument to a module identifier.
```

Here, the algebraic datatype of the anonymous argument cannot be referred and is yet necessary for the resulting signature: there is not signature *avoiding* this type. The error message suggests a solution: naming the module argument to make its type components available for the rest of the program.

Anonymous `open` OCaml also offers anonymous **`open`** [Li and Yallop, 2017], a feature that is discussed in Section 6.1.1. An *opened* module must not appear in the resulting signature (it would otherwise escape its scope):

```
module M = struct
  open (struct type t end)
  let x : t option = None
end
```

```
Error: The type t/4046 introduced by this open appears in the signature
Line 5, characters 6-7:
The value x has no valid type if t/4046 is hidden
```

There is no signature *avoiding* the type `t`.

Local module OCaml allows an interleaving of value and module definitions via *local modules*, i.e., modules defined locally inside a value definition (see Section 6.2.1)

```
let f (x: int) =
  let module X = (struct type t = A of int | B end) in
  X.A(x)
```

```
Error: This expression has type X.t but an expression was expected of type 'a
       The type constructor X.t would escape its scope
```

Overall, signature avoidance can appear as long as there are ways to hide abstract types from the typing environment, creating dangling dependencies. There is therefore a mismatch, illustrated in Figure 5, between the expressiveness of the module and signature languages: the *reachable space* of possible module expressions is larger than the *describable space* of signatures: some modules simply cannot be described by a signature.

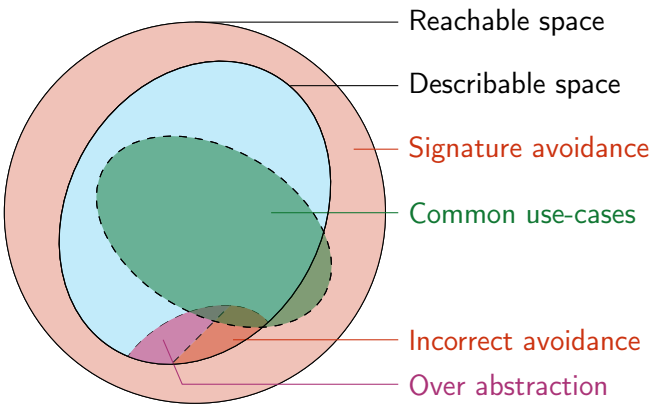


Figure 5: A representation of the mismatch between the *reachable space* of module expressions (outer-most circle) and the *describable space* of signatures (inner ellipse). The common use-cases of OCaml are mainly within the area where the typechecker behaves correctly. In some cases, the current OCaml typechecker can lose type-equalities while still being in the describable space. This may lead to (1) producing a signature where some type fields are unnecessarily made abstract (over-abstraction) or (2) failing at inferring the signature (incorrect avoidance).

2.3.2 Strategies

There are three main strategies to handle the avoidance problem:

1. Preventing the loss of abstract types. For instance, naming all module expressions make all abstract types always accessible and the avoidance problem disappears. However, it greatly limits the usability of the module calculus and prevents a fine-grained management of types: it becomes impossible to hide intermediary module constructions without explicit ascriptions.

2. Trying to rewrite the signature to avoid the out-of-scope types. This might require deep rewrites inside the signature, to abstract type components, as done by Crary [2020]. OCaml uses an incomplete (undocumented) heuristic that can lead to *loss of type-sharing*: type equalities between in-scope types might be lost. We describe this heuristic below.

3. Extend the signature syntax to account for the existence of out-of-scope types, as done in SML'90 [Milner et al., 1997] and in [Dreyer et al., 2003; Russo, 2004, 1996; Rossberg et al., 2014; Harper and Stone, 2000]. From a formalization point of view, the extension of syntax forces a specification *by elaboration*, where signatures are translated in a more expressive language, more or less distant from the source one.

The rewriting heuristic of OCaml When a signature mentions an out-of-scope type, say `t` for instance, OCaml uses the following heuristic:

1. If the type appears in a type field of the form `type u = t` that is in a strictly positive position, the field is transformed into an abstract type field `type u`

2. If the type appears in a module type field, the whole module type field is transformed into an abstract module type, which is actually a bug[9] .

3. If the type appears in any other position (value field, non-strictly positive type field, etc), a type-error is produced.

[9]Researching this behavior led us to uncover this bug and open the issue OCaml #10491

Loss of type-sharing While simple, this heuristic has some surprising behavior. Let us consider the following code:

```
module F = functor (Y:sig type t end) → struct type u = Y.t type v = Y.t end
module M = F(struct type t = A | B end)
```

Here, the signature of `X` mentions an out-of-scope type `t` that should be avoided. OCaml make all type declarations that mention `t` abstract, giving the following signature [10]:

```
module F : functor (Y:sig type t end) → sig type u = Y.t type v = Y.t end
module M : sig type u type v end (* OCaml *)
```

Here, there is a loss of type-sharing: the type `u` and `v` are no longer equal! It can lead to unexpected errors later in the program:

```
let f : M.u → M.v = fun x → x
```

```
Error: This expression has type M.u but an expression was expected of type M.v
```

Yet, a better signature was possible if instead of abstracting all type declarations mentioning `t`, the typechecker would recognize `u` as an available in scope alias that can be used to replace `t`:

```
module M : sig type u type v = u end
```

Inside the body of the functor, `Y.t` and `u` refer to the same type, and using one or the other in the definition of `v` should not change the result. Yet, writing `type v = u` instead of `type v = Y.t` would change the signature inferred by OCaml: the avoidance resolution heuristic would not rewrite a declaration that does not use out-of-scope types. Overall, the resolution mechanism is quite brittle: the typechecker can lose type equalities, and this depends on the choice of (equivalent) type aliases in the definitions.

> **Type-preserving anchoring**: all type equalities should be preserved: i.e., types that are considered equal before a projection and that remain in scope should still be considered equal after a projection. Type definitions should not be abstracted away silently, but only by an explicit opaque ascription. Resolution of avoidance should not depend on the choices of aliases in the definitions.

2.3.3 Avoidance with applicative functors

Introducing applicative functors in the mix adds a new layer of complexity. While it is possible to define a type-preserving resolution of avoidance that supports applicative functors, we believe that it should be restricted to rely on module equivalence. To support that point, we present the difficulty of designing such resolution through examples, but, more importantly, we show why we believe that it would be impractical.

Change of domain The abstract type of applicative functors can be thought of as type functions over a certain *domain*: the signature of the parameter of the functor.

```
(struct
  module F (Y:S) = struct type t end
  module R = struct
    module F1 (Y:S1) = struct type t = F(Y).t end
    module F2 (Y:S2) = struct type t = F(Y).t end
```

[10] In OCaml 4.14, the `-short-path` option can give a misleading output, writing the following signature for the functor: `module F : functor (Y:sig type t end) → sig type u = Y.t type v = u end`. It is misleading, because internally the type equality is `type v = Y.t` and not `type v = u`. Therefore, the avoidance algorithm will in fact rewrite the type field, making it abstract `type v` in the resulting signature after the application.

```
    module X : S' = ...
    type t = F(X).t
  end
end).R
```

Using the abstraction heuristic of OCaml, we would get:

```
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t end
  module X : S'
  type t
end
```

All type equalities have been lost, while some could be kept! To preserve type-sharing, one might be tempted to abstract the type-definition inside `F1` and rewrite all paths using `F` with `F1` instead. However, it might not be possible, as `F` and `F1` do not have the same parameter signature (resp. `S` and `S1`). To check if it is correct, one would have to browse the rest of the signature to find if `F` is only "used" on a subset of its domain, which makes the resolution of avoidance non-local. If `S2` and `S'` happen to be subtypes of `S1`, then the following signature is correct:

```
(* if S2 < S1 and S' < S1 *)
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t = F1(Y).t end
  module X : S'
  type t = F1(X).t
end
```

If, instead, `S1` and `S2` are incompatible, i.e., no signature is subtype of both, then it would not loose type-sharing to abstract the type fields inside the body of both functors:

```
(* if there is no S such that S < S1 and S < S2 *)
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t end
  module X : S'
  type t = F1(X).t
end
```

Change of arity In addition to the domain, the arity of abstract types could be changed. We modify the previous example by adding a dummy parameter to `F1`:

```
(struct
  module F (Y:S) = struct type t end
  module R = struct
    module F1 (Y:S1) (Arg: sig end) = struct type t = F(Y).t end
    module X : S' = ...
    module A = struct end
    type t = F(X).t
  end
end).R
```

Again, the abstraction heuristic of OCaml would abstract all type fields mentioning `F`. However, in order to use `F1` in place of `F`, we would have to choose a second argument to the application. This choice would be completely arbitrary and very surprising to the user:

```
sig
  module F1 (Y:S1) (Arg: sig end) : sig type t end
```

```
  module X : S'
  module A : sig end
  type t = F1(X)(A).t
end
```

No invention of functor application Both the examples above constitute cases where the typechecker would try to solve signature avoidance by creating new functor applications "out of thin air", just to refer to types that have lost their original path. While sound, this would be very surprising to the user, if not misleading. Let us consider this final example, where the projection renders the `List` functor inaccessible.

```
module M = (struct
  module type T = ...
  module Lists (Y: T) : sig type t (* other fields ... *) end = (* ... *)
  module R = struct
    module Sets (E: T) = struct
      (* ... *)
      module ToList = struct
        type elist = Lists(E).t
        (* extraction from sets to lists ... *)
      end
    end
    module X = (* ... *)
    type t = Lists(X)
  end).R
```

An advanced anchoring algorithm could find the following signature:

```
module M : sig
  module Sets(E:T) : sig
    (* ... *)
    module ToList : sig
      type elist
      (* .. *)
    end
  end
  module X : (* ... *)
  type t = Sets(X).ToList.t
end
```

Here, it *suggests* the computation of the module `Sets(X)`, while this application was no present in the original code. We take the stance that such rewrites, while doable, would be confusing for the user and very hard to predict.

Restricting anchoring Overall, types inside functor applications cannot be considered "independently" of the functor they were defined in. So if a functor `F` is lost by projection, when should anchoring *not produce* a type error? Here, we suggest an answer: if a functor `F'` *equivalent* to `F` is still in scope! We reuse the notion of module equivalence explored in Section 2.2.4: `F'` must have a *transparent signature* of the form `(= F < ...)`. This gives a simple and predictable criterion, which we propose for the design of anchoring:

> **P** **Resolution of avoidance based on module equivalence**: a path `F(X).t` can avoid `F` if a module equivalent to `F` is still in scope, and can avoid `X` if a module equivalent to `X` is still in scope.

We believe that the simplicity of this criterion is a key design choice for usability: it reduces the problem of higher-order avoidance to a first-order problem at the module level.

2.3.4 Signature avoidance in practice

Naming modules OCaml users usually get around signature avoidance errors by explicitly naming modules before using them, which adds *always-accessible* type definitions. The module syntax of OCaml actually encourages this approach by limiting the places where inlined, anonymous modules can be used. In particular, projection on an anonymous module is forbidden. It is however cumbersome, as it prevents some concise code patterns and forces to expose type definitions.

Interface files Moreover, in practice, OCaml developers often have interface files (with the extension `.mli`) that behave as a file-wide ascription. This means that signature avoidance is actually a problem that occurs during typechecking of intermediary module expressions, before the final ascription forces a given signature. Therefore, we advocate for a mechanism that does not fail when sub-expressions cause signature avoidance:

> **Delayed avoidance**: the module system should tolerate and support signatures with out-of-scope types for intermediary module expressions

2.4 Module-level abstraction

In this section we present *module-level abstraction*. OCaml features a unique construct in the ML-family: *abstract signatures*. They offer module-level ascription and polymorphism, but come with their own set of challenges. Abstract signatures are both one of the most esoteric and under-appreciated features of the OCaml module system, which is the only language of the ML family to have such feature.

Just as signatures can contain abstract type declarations of the form `type t` (without a definition), they can contain abstract *module-type* declarations, of the form `module type T` (without a definition).

In Section 2.4.1, we start by introducing abstract signatures through examples. In Section 2.4.2, we discuss some issues associated with the feature. We argue that it has surprising behaviors and, in its current *unrestricted* form, it is actually *too powerful for its own good*. In Section 2.4.3, we propose a restriction to make abstract signatures *predicative* which, by decreasing their expressiveness, actually makes them more usable.

2.4.1 Abstract signatures

We introduce abstract signatures via two on-going examples. Fundamentally, they will not be surprising to readers used to relying on abstraction to protect invariants and factor out code. Their specificity lies in the fact that they are at the module level, and therefore require projects with a certain size and a strong emphasis on modularity to be justified.

Module sealing

Let's consider the following scenario. Two modules providing an implementation of UDP (`UDP1` and `UDP2`) are developed with different design trade-offs. They both implement a signature with basic send and receive operations. Then, functors are added as layers on top: taking a UDP library as input, they return another UDP library as an output.

- `Reliable` adds sequence numbers to the packets and re-sends missing packets;
- `CongestionControl` tracks the rate of missing packets to adapt the throughput to network congestion situations;
- `Encryption` encrypts the content of all messages.

The resulting signature might look something like:

```
module type UDPLib = sig
  module type UDP = sig
    val send : string → unit
    val get : unit → string
    (* ... *)
  end

  module UDP1 : UDP
  module UDP2 : UDP

  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP → UDP
end
```

However, a project might need different combinations of the basic libraries and functors, while requiring that **all** combinations use encryption. To enforce this, the solution is to use the module-level *sealing* of abstract signatures. In practice, the signature of the whole library `UDPLib`, containing implementations and functors (typically, its `.mli` file) is rewritten to abstract the `UDP` interface, except for the output of the `Encryption` functor.

```
module type UDPLib = sig
  module type UDP (* abstract! *)

  module UDP1 : UDP
  module UDP2 : UDP

  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP →
    sig
      val send : string → unit
      val get : unit → string
      (* ... *)
    end
end
```

Just as type abstraction, signature abstraction can be used to enforce certain code patterns: users of `UDPLib` will only be able to use the module after calling the `Encryption` functor, and yet they have the freedom to choose between different implementations and features:

```
module UDPKeyHandshake = Encryption(Reliable(UDP1))
module UDPVideoStream  = Encryption(CongestionControl(UDP2))
```

Module polymorphism

Another use is to introduce polymorphism at the module level. Just as polymorphic functions can be used to factor code, module-level polymorphic functors can be used to factor module expressions. If a code happens to often feature functor applications of the form `Hashtbl.Make(F(X))` or `Set.Make(F(X))`, one can define the `MakeApply` functor as follows:

```
(* Factorizing common expressions *)
module type Type = sig module type T end
module MakeApply
  (A:Type) (X: A.T)
  (B:Type) (F: A.T → B.T)
  (C:Type) (H: sig module Make : B.T → C.T end) = H.Make(F(X))
```

Downstream the code is rewritten using `MakeApply`. Right now, the verbosity of such example would probably be a deal-breaker. Ignoring the verbosity, this can be useful for maintenance: by channeling all applications through `MakeApply`, only one place needs to be updated if the arity or order of arguments is changed. Similarly, if several functors expect a *constant argument* containing – for instance – global variables, an `ApplyGv` functor can be defined to always provide the right second argument, which can even later be hidden from the user of `ApplyGv`:

```
(* Constant argument *)
module Gv : GlobalVars
module ApplyGv (Y : sig module type A module type B end)
    (F : Y.A → GlobalVars → Y.B)(X : Y.A) = F(X)(Gv)
```

Downstream, code featuring `F(X)(GlobalVars)` is rewritten into `ApplyGv(...)(F)(X)`. Then, the programmer can hide the `GlobalVars` module while letting users use `ApplyGv`, ensuring that global variables are not modified in uncontrolled ways by certain part of the program.

Finally, polymorphism can also be used by a developer to prevent unwanted dependencies on implementation details. If the body of a functor uses an argument with a submodule `X`, but actually does not depend on the content of `S`, abstracting it is a "good practice":

```
module F (Arg : sig ... module X : S ... end) =
struct (* the code can depend on the content of S *) end

module F' (Y: sig module type S end)
  (Arg : sig ... module X : Y.S ... end ) =
struct (* the code cannot depend on the content of S *) end
```

2.4.2 Challenges of abstract signatures

In this subsection we explore the issues posed by abstract signatures as currently implemented in OCaml.

Variant interpretation

The challenge for understanding (and implementing) abstract signatures lies more in the meaning of the module-level *polymorphism* that they offer than the module level sealing, the latter being pretty straightforward. More specifically, the crux lies in the meaning of the *instantiation* of an abstract module-type variable `A` by some other signature `S`, that happens when a polymorphic functor is applied. The substitution-based intuition ("*replacing all occurences of* `A` *by* `S`") has some surprising behaviors when the signature `S` contains abstract types, as they have a *variant* interpretation: an abstract type in positive position indicates sealing, while an abstract type in negative position indicates polymorphism. Said differently, abstract signatures viewed as higher order types are not covariant nor contra-variant.

Therefore, when instantiating an abstract signature with a signature that has abstract fields, the user must be aware of this, and mentally infer the meaning of the resulting signature. To illustrate how it can be confusing, let's revisit the first motivating example and let's assume that the developer actually wants to expose part of the interface of the raw `UDP` libraries. One might be tempted to instantiate `UDP` with something along the following lines[11]:

```
module type UDPLib_expose = sig
  include UDPLib with module type UDP =
    sig
      module type UNSAFE
      module Unsafe : UNSAFE     (* this part remains abstract *)
```

[11] We use the construct **`with module type`** for the instantiation, which is introduced in Section 6.2.2.

```
      module Safe : sig ... end  (* this part is exposed *)
    end
end
```

Which ex

```
module type UDPLib_expose =  sig
  module type UDP = sig
      module type UNSAFE
      module Unsafe : UNSAFE
      module Safe : sig ... end  end
  module UDP1 : UDP
  module UDP2 : UDP
  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP → sig val send : string → unit (* ... *) end
end
```

However, the variant interpretation of this signature in the negative positions produces a counter-intuitive result. If we expand the signature of the argument for the functor 'Reliable' (for instance) we see:

```
module Reliable :
sig
  module type UNSAFE
  module Unsafe : UNSAFE
  module Safe : sig ... end
end → UDP
```

This forces the functor to be *polymorphic* in the underlying implementation of `UNSAFE`. By contrast, when the whole `UDP` signature is abstract, the functor does not have to be polymorphic and might depend on the internal details. Therefore, exposing part of `UDP` while keeping some abstract completely changes the meaning of the signature. If the user still wants to expose part of the signature without creating unwanted polymorphism, another pattern can be used. A `shared` unsafe core is defined separately and `UDP` is rewritten to refer to it:

```
module type UDPLib_expose' = sig
  module type UNSAFE
  include UDPLib with module type UDP = sig
    module Unsafe : UNSAFE
    module Safe : sig ... end
  end
end
```

Doing so, the instantiated signature does not contain abstract fields and therefore its variant reinterpretation will not introduce unwanted polymorphism.

Impredicativity

Abstract module types are impredicative: a signature containing an abstract signature can be instantiated by itself. One can trick the subtyping algorithm into an infinite loop of instantiating an abstract signature by itself, as shown by Rossberg[12], adapting an example from Harper and Lillibridge [1994]:

```
module type T = sig
  module type A
  module F : A → sig end
end

```

[12] See https://sympa.inria.fr/sympa/arc/caml-list/1999-07/msg00027.html

```
module type I = sig
  module type A
  module F : (T with module type A = A) → sig end
end

module type J = (T with module type A = I)

module Loop (X:J) = (X:I)
```

During the typechecking of the functor `Loop`, a subtyping check is made between `J` and `I`, i.e., between `T[A <- I]` and `I`. This leads to instantiating `A` by `I` in the right-hand side. When comparing the two functor declarations, the subtyping between the two arguments is again (by contravariance) the subtyping between `T[A <- I]` and `I`: the typechecker is thrown in a loop.

Impredicativity also allows type-checking of (non-terminating) programs with an absurd type, as shown by the encoding of Girard's paradox by Leo White[13].

Module-level-sharing

Abstract signatures require module-level sharing, like the transparent signatures presented in Section 2.2.4. To see why, let us consider the following two functors:

```
module F1 (Y: sig module type A module X : A end) = Y.X
module F2 (Y: sig module type A module X : A end) = (Y.X : Y.A)
```

Currently, both are given the same type:

```
module F1 (Y: sig module type A module X : A end) : Y.A
module F2 (Y: sig module type A module X : A end) : Y.A
```

However, we would expect the body of `F1` to preserve type-sharing with its argument, while the `F2` uses an opaque ascription and should indeed loose type-sharing. With transparent signatures, we would get:

```
(* Currently, both are given the same type: *)
module F1 (Y: sig module type A module X : A end) : (= Y.X < Y.A)
module F2 (Y: sig module type A module X : A end) : Y.A
```

Other issues

The implementation of abstract signatures has a number of issues besides the theoretical known ones mentioned above. We uncovered new ones, notably: OCaml #12204 and OCaml #10491.

2.4.3 Simple abstract signatures

In this section we present a restriction for abstract signatures that seems to cover all useful cases. The main criticism we make of the OCaml approach is that it is actually *too expressive for its own good*. Having impredicative instantiation with variant reinterpretation is hard to track for the user and interacts in very subtle ways with other features of the module system. To address this, we take the opposite stance and propose to make the system actually *predicative*: we restrict the set of signatures that can be used to instantiate an abstract signature. This also indirectly addresses the complexity of the variant reinterpretation.

[13] See https://github.com/lpw25/girards-paradox/tree/master

> ➕ M **Predicative abstract signatures**: we propose to restrict the instantiation of abstract signatures by *simple signatures*: signatures that may contain abstract type fields but no abstract module-type fields.

Expressivity One might wonder how restrictive this proposal is. Specifically, if we consider a simple polymorphic functor as:

```
module Apply (Y : sig module type A end) (F : Y.A → Y.A)(X : Y.A) = F(X)
```

The following partial application would be rejected:

```
(* Rejected as A would be instantiated by
  'sig module type B module X : B → B end' *)
module Apply' = Apply(struct
  module type A = sig module type B module X : B → B end
end)
```

However, this could be circumvented by eta-expanding, thus expliciting module type parameters, and instantiating only a simple signature:

```
(* Accepted as A is instantiated by a signature with no abstract fields *)
module Apply'' = functor (Y:sig module type B end) →
  Apply(struct
    module type A = sig module type B = Y.B module X : B → B end
  end)
```

Module-type arguments for functors A key aspect of abstract module types that reduces their usability is the fact that signatures have to be given as part of a module. Instead, we propose *module-type* arguments for functors. In practice, they could be indicated by using brackets instead of parenthesis, and interleaved with normal module arguments, as in this example:

```
(* At definition *)
module MakeApply
    [A] (X:A)
    [B] (F: A → B)
    [C] (H : sig module Make : B → C end)
  = H.Make(F(X))

module ApplyGv
    [A] [B] (F:A → GlobalVars → B) (X:A)
  = F(X)(Gv)

(* At the call site *)
module M1 = MakeApply
    [T] (X)
    [Hashtbl.HashedType] (F)
    [Hashtbl.S] (Hashtbl)

module M2 = ApplyGv [A] [B] (F) (X)
```

Technically, this is not just syntactic sugar for anonymous parameters due to the fact that OCaml relies on names for applicativity of functors. Similarly, type arguments for functors could also be useful. Further work is needed to see if they are worth the effort. We conjecture that the introduction of *modular implicits* (see Section 6.2.1) could bring more use-cases for those features. Finally, usability of abstract signatures could even be improved with some

form of inference at call sites. Further work is needed to understand to what extent this could be done, and how useful it would be.

Chapter 3

The ML source system

In this short technical chapter we present our OCaml-like language that we study in the rest of this thesis. We refer to it as the *source* language. We discuss the syntax, some syntactic mechanisms, and the semantics of the language in this chapter. We present two type systems for this language in the next two chapters (Chapter 4,Chapter 5).

Overview In Section 3.1, we present the syntax and discuss technical choices, syntactic sugar, and other practical aspects. In Section 3.2 we define a type-erasing semantics on the source syntax.

3.1 Syntax

The source syntax is given in Figure 6, and discussed below. The language is the combination of three main parts: *paths and identifiers*, *module and signatures* and the *core language*. We start with some minor notational choices:

- **Fonts**: module-related meta-variables use *typewriter uppercase letters*, `M`, `S`, etc., while lowercase letters are used for expressions and types of the core language. Lists are written with an overhead bar: $\overline{\texttt{D}}$ is a list of `D`. Identifiers I and paths P use a standard (mathematical) font.
- **Structures and components**: we separate declarations `D` from signatures `S` and, respectively, bindings `B` from module expressions `M`. In the literature, declarations and signatures (and bindings and modules) are sometimes merged in the same syntactic category.
- **Functor parameters** Y are α-convertible, i.e., can be freely renamed. The other identifiers (X, T, x, and t) cannot be renamed, as they play the role of both internal and external names.
- **Code inserts** to distinguish them from OCaml (on the left-hand side), the code insert in our source language (on the right-hand side) have a blue background, as in:

```
module M = struct type t = int end
```

```
module M = struct type t = int end
```

Self-references In order to simplify the treatment of scoping and shadowing, we introduce *self-references*, ranged over by letter A, in both structures and signatures. They are used to refer to the current object; their binding occurrence appears as a subscript to the structure or signature they belong to ($\texttt{struct}_A \ldots \texttt{end}, \texttt{sig}_A \ldots \texttt{end}$). They are α-convertible. Importantly, they **do not serve as a way to define recursive structures** or define forward references. They will serve to make structures and signatures behave as *telescopes*, where each field can access **previously** defined ones. We explain how they help treat shadowing in Section 3.1.1. OCaml could be easily translated to fit into this syntax, by a simple pass that introduces a fresh self-reference to every structure and structural signature present in the code.

Prefixes In order to have a uniform treatment of accesses to local and non-local variables, we use *prefixes*, written with the letter Q, to range over either a path P or a self-reference A.

Paths and Prefixes

$P ::= Q.X$	(Qualified module)
$\mid Y$	(Module Parameter)
$\mid P(P)$	(Applicative application)
$Q ::= A \mid P$	(Prefix)

Identifiers

$I ::= x$	(Variable)
$\mid t$	(Type)
$\mid X$	(Module)
$\mid T$	(Module type)

Module Expressions

$\mathtt{M} ::= \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}$	(Structure)
$\mid P$	(Path)
$\mid (Y : \mathtt{S}) \to \mathtt{M}$	(Applicative functor)
$\mid () \to \mathtt{M}$	(Generative functor)
$\mid P()$	(Generative application)
$\mid \mathtt{M}.X$	(Anonymous projection)
$\mid (P : \mathtt{S})$	(Ascription)

Signatures

$\mathtt{S} ::= \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}$	(Structural signature)
$\mid (= P < \mathtt{S})$	(Transparent signature)
$\mid (Y : \mathtt{S}) \to \mathtt{S}$	(Applicative functor)
$\mid () \to \mathtt{S}$	(Generative functor)
$\mid Q.T$	(Module type)

Bindings

$\mathtt{B} ::= \mathtt{let}\, x = \mathtt{e}$	(Value)
$\mid \mathtt{type}\, t = \mathtt{u}$	(Type)
$\mid \mathtt{module}\, X = \mathtt{M}$	(Module)
$\mid \mathtt{module\ type}\ T = \mathtt{S}$	(Module type)

Declarations

$\mathtt{D} ::= \mathtt{val}\, x : \mathtt{u}$	(Value)
$\mid \mathtt{type}\, t = \mathtt{u}$	(Type)
$\mid \mathtt{module}\, X : \mathtt{S}$	(Module)
$\mid \mathtt{module\ type}\ T = \mathtt{S}$	(Module type)

Core language expressions

$\mathtt{e} ::= \ldots$	(Other constructs)
$\mid Q.x$	(Qualified value)

Core language types

$\mathtt{u} ::= \ldots$	(Other constructs)
$\mid Q.t$	(Qualified type)

Figure 6: Syntax of our OCaml-like language

Abstract types We use a convention to represent abstract type fields: they are type fields pointing to themselves, of the form $\texttt{type}\, t = A.t$ where A is the self-reference of the current structure (which is often grayed out to emphasize that there is no actual definition). It is a way to remove the need to distinguish between abstract and concrete type fields in the syntax, and simplify the treatment of strengthening in Chapter 5.

Core language We leave abstract a core language of expressions $\texttt{e}$ and types $\texttt{u}$. This language can be thought of as ML, but could also contain exotic features. We only extend the language of terms with *qualified values* $Q.x$ that are values bound by the module level, and the language of types with *qualified types* $Q.t$ that are types bound by the module level. These are the only ways for the core level to access the module level.

Functors As in OCaml, we syntactically distinguish applicative and generative functors: generative functors take a special unit argument () as input. The unit argument () is not the same as an empty structure $\texttt{struct}\ \ \texttt{end}$. We find the effect-suggesting syntax of taking a unit argument fitting, as calling a generative functor can indeed produce effects. Our grammar features *unary* applicative functors and *nullary* generative functors. A unary generative functor can be obtained as an applicative functor returning a generative one via the currying notation:

$$(Y : \texttt{S})\ () \to \texttt{M} \triangleq (Y : \texttt{S}) \to () \to \texttt{M}$$

While n-ary applicative functors are straightforward, one might wonder if n-ary generative functors require a unit argument between every parameter. Actually, the () acts as a *generative barrier* and can be placed to control the sharing between partial applications:

$$(Y_1 : \texttt{S}_1)(Y_2 : \texttt{S}_2)() \to M$$

is *fully generative* (every instance is new), while

$$(Y_1 : \texttt{S}_1)()(Y_2 : \texttt{S}_2) \to M$$

is generative with regard to the first argument and applicative with regard to the second one.

Projectibility Choosing (1) whether projection is allowed on any module expression or only on a restricted subset, and (2) how the core language can refer to values and types of modules is an important design choice in ML systems, coined *projectibility* by Dreyer et al. [2003]. Contrary to *F-ing* (Rossberg et al. [2014]), but following Leroy [1995] and Russo [2004] (and others), we chose to use a *syntactic* notion of path.

- We allow projection on any module expression, but we restrict functor applications and ascriptions to paths. OCaml does the opposite, mainly to prevent code patterns prone to triggering signature avoidance. Our choice is more general, as we can define a *let* construct for modules using the following syntactic sugar:

  $$\mathsf{let}\ X = \texttt{M}\ \mathsf{in}\ \texttt{M}' \triangleq (\texttt{struct}_A\ \texttt{module}\ X = \texttt{M}\ \ \texttt{module}\ \mathsf{Res} = \texttt{M}'\ \texttt{end}).\mathsf{Res}$$

  Using this construct, we easily get functor application and ascription on arbitrary module expressions as syntactic sugar:

  $$\begin{aligned} \texttt{M}(\texttt{M}') &\triangleq \mathsf{let}\ F = \texttt{M}\ \mathsf{in}\ \mathsf{let}\ X = \texttt{M}'\ \mathsf{in}\ F(X) \\ \texttt{M}() &\triangleq \mathsf{let}\ G = \texttt{M}\ \mathsf{in}\ G() \\ (\texttt{M} : \texttt{S}) &\triangleq \mathsf{let}\ X = \texttt{M}\ \mathsf{in}\ (X : \texttt{S}) \end{aligned}$$

  By contrast, the OCaml choice forces encoding projection $\texttt{M}.X$ as an anonymous functor call $((Y : \texttt{S}) \to Y.X)\,\texttt{M}$. This requires an explicit signature annotation on the argument and thus cannot be seen as syntactic sugar.

- A qualified access inside a generative functor application, of the form $G().t$, is syntactically ill-formed, as paths do not contain the unit argument $()$. By contrast, a qualified access inside an applicative functor application $F(X).t$ is permitted.
- A qualified access inside a module type, which would be of the form $Q.T.t$, is syntactically ill-formed, as paths do not contain module type identifiers T.
- We only provide opaque ascription in module expressions, as concrete ascription is given by the following syntactic sugar: $(P < \mathtt{S}) \triangleq (P : (= P < \mathtt{S}))$

As both path and module expressions feature a projection dot, the grammar is slightly ambiguous. However, this is not a problem as we see paths as a subset of module expressions. In particular, we only consider the projection dot of module expressions in the typing rules.

Transparent ascription Transparent ascription as a module expression, as available in SML, can be encoded using either a normal ascription with a transparent signature, or as a anonymous functor call:

$$\begin{array}{ll} \text{either} & (\mathtt{M} < \mathtt{S}) \triangleq \mathsf{let}\ X = \mathtt{M}\ \mathsf{in}\ (X : (= X < \mathtt{S})) \\ \text{or} & (\mathtt{M} < \mathtt{S}) \triangleq \mathsf{let}\ F = ((Y : \mathtt{S}) \to Y)\ \mathsf{in}\ F(M) \end{array}$$

Program Formally, we can define the top-level of a file as the body of a generative functor

Example 3.1.1. *We compare side by side an* OCaml `.ml` *file (left-hand side) and the same program in our source language (right-hand side):*

```
1  (* code.ml *)
2  module type T = sig
3    type t
4    val x : t
5  end
6  module X : T = struct
7    type t = int
8    let x : t = 42
9  end
10 module F (Y:T) : T = Y
11 module X' = F(X)
```

```
1  () → sig_A
2   module type T = sig_B
3     type t = B.t
4     val x : B.t
5   end
6   module X = (sig_C
7     type t = int
8     let x = (42 : C.t)
9    end : A.T)
10  module F = (Y : A.T) → (Y : A.T)
11  module X' = A.F(A.X)
12 end
```

The self-references are used to disambiguate the identifiers. We often gray them out for readability. The abstract type field at line 3 is written as a type pointing to itself. The whole program is the body of an applicative functor.

3.1.1 Name-spaces

Scopes The structure of paths defines *scopes*: names are accessible only in some parts of the code. The two constructs that introduce a new scope in signatures are generative functors and module types declarations:

$$() \to \mathtt{S} \qquad\qquad \mathtt{module\ type}\ T = \mathtt{S}$$

The fields defined in `S` introduce names that are not accessible outside of `S`, as there is no path with a unit argument and no path with a projection out of a module type name. Similarly for module expressions, generative functors and module type bindings create new scopes. By

contrast, module declarations and applicative functors introduce sub-scopes that are still part of the same enclosing scope. If we consider:

$$\texttt{module}\, X : \texttt{sig}_A\ \texttt{type}\ t = \ldots\ \texttt{end} \qquad\qquad (Y : \texttt{S}_a) \to \texttt{sig}_A\ \texttt{type}\ t = \ldots\ \texttt{end}$$

the name t is accessible outside of the signature, via a qualified access with a path.

3.1.2 Shadowing

There are two main variants of shadowing: *direct* and *local*. Direct shadowing occurs when two bindings are made with the same name in the same structure or the same structural signature. OCaml allows direct shadowing of value fields in structures, which is used by some coding patterns, but disallows direct shadowing of type fields[1]. In the type systems of both Chapter 4 and Chapter 5, we disallow direct shadowing. Local shadowing occurs when a binding made inside a structure uses the same name as a binding coming from an enclosing structure. In our language, we use self-references to disambiguate between the two bindings. Let us consider the following example in OCaml (on the left-hand side) and our language (on the right-hand side):

```
1 module M = struct
2   type t = int
3   module X = struct
4     type t = bool
5     val x : t = true
6   end
7   type u = t * X.t
8 end
```

```
1 module M = struct_A
2   type t = int
3   module X = struct_B
4     type t = bool
5     let x : B.t = true
6   end
7   type u = A.t × A.X.t
8 end
```

Using self-references, we can distinguish $A.t$ and $B.t$: they are considered as different qualified types. In OCaml, at line 5, the name `t` refers to the local definition at line 4, which has shadowed the declaration at line 2. This shadowing ends when we exit the submodule, at line 6, then both declarations become accessible simultaneously. Internally, OCaml generates fresh identifiers attached to every name for disambiguation. This serves more-or-less the same purpose as self-references, but in an effectful way that would be hard to model formally. By contrast, self-references act as a normal name binder attached to the structure (or signature), they can be freely renamed (α-convertibility). Additionally, self-references also help the treatment of strengthening in Chapter 5. We believe that overall, they simplify the formal presentation at the cost of being more verbose.

3.2 Semantics

ML modules have (surprisingly) very simple semantics. Basically, at runtime, all types are erased (the semantic is said to be *type-erasing*) and modules are elaborated away into the untyped core-language constructs (using functions and records). The type-erasing semantics ensures that safety guarantees of the type system are obtained at no cost at runtime.

Core language We assume that the untyped core language supports simple untyped functions, records, and unit arguments. Formally, we suppose that it extends the following calculus:

$$e ::= x \mid () \mid \lambda x.e \mid e\, e \mid \{\overline{\ell_i = x_i}\} \mid e.\ell \mid e \mathbin{@} e \mid \ldots \qquad \text{(Terms)}$$

$$v ::= x \mid () \mid \lambda x.e \mid \{\overline{\ell_i = v_i}\} \mid \ldots \qquad \text{(Values)}$$

[1] **include** and **open** have a special treatment: the declarations or bindings imported by them can be shadowed, even type fields, but they cannot be used to shadow neither a type or value field.

We denote record concatenation with the symbol @. For the semantics, we define a notion of evaluation contexts $C[\bullet]$ as:

$$C[\bullet] ::= \bullet\, e \mid v\, \bullet \mid \left\{\overline{\ell = v}, \ell_i = \bullet, \ldots\right\}$$

The small-steps semantic judgment, written $e \rightsquigarrow e'$, is given by the following rules (where $\#$ denotes disjointedness of the lists of labels):

$$(\lambda x.e)\, v \rightsquigarrow e[x \mapsto v] \qquad \{\ldots, \ell = v, \ldots\}.\ell \rightsquigarrow v$$

$$\frac{\overline{\ell_1} \# \overline{\ell_2}}{\left\{\overline{\ell_1 = v_1}\right\} @ \left\{\overline{\ell_2 = v_2}\right\} \rightsquigarrow \left\{\overline{\ell_1 = v_1}, \overline{\ell_2 = v_2}\right\}} \qquad \frac{e \rightsquigarrow e'}{C[e] \rightsquigarrow C[e']}$$

Type erasure We can then define an erasing operation $\lfloor\cdot\rfloor$ that translates modules and paths into terms of this untyped calculus. We start with two technical details:

1. We assume a collection of labels indexed by identifiers, i.e., ℓ_I is the unique label associated with the identifier I
2. We assume that variables of the core language can be extended to contain a collection of variables of the form $x_{A,I}$ for any self-reference A and identifier I and contain a collection of variables or the form x_Y for any functor parameter Y.

We define the erasing translation. Elaboration of paths is defined by induction as follows:

$$\lfloor A.X \rfloor = x_{A,X} \qquad \lfloor P.X \rfloor = \lfloor P \rfloor.\ell_X \qquad \lfloor Y \rfloor = x_Y \qquad \lfloor P(P') \rfloor = \lfloor P \rfloor\, (\lfloor P' \rfloor)$$

Then the elaboration of module expressions and bindings is defined inductively as follows:

$$\lfloor (P : \mathtt{S}) \rfloor = \lfloor P \rfloor \qquad \lfloor \mathtt{M}.X \rfloor = \lfloor \mathtt{M} \rfloor.\ell_X \qquad \lfloor \mathtt{struct}_A\ \overline{\mathtt{D}}\ \mathtt{end} \rfloor = \lfloor \overline{\mathtt{D}} \rfloor_A \qquad \lfloor () \to \mathtt{M} \rfloor = \lambda x.\lfloor \mathtt{M} \rfloor$$

$$\lfloor (Y : \mathtt{S}) \to \mathtt{M} \rfloor = \lambda x_Y.\lfloor \mathtt{M} \rfloor \qquad \lfloor P() \rfloor = \lfloor P \rfloor\, ()$$

$$\lfloor \mathtt{let}\, x = \mathtt{e}, \overline{\mathtt{D}} \rfloor_A = (\lambda x'_{A,x}.\left\{\ell_x = x'_{A,x}\right\} @ \lfloor \overline{\mathtt{D}} \rfloor_A)\, \lfloor e \rfloor \qquad \lfloor \mathtt{type}\, t = \mathtt{u}, \overline{\mathtt{D}} \rfloor_A = \lfloor \overline{\mathtt{D}} \rfloor_A \qquad \lfloor \varnothing \rfloor_A = \{\}$$

$$\lfloor \mathtt{module}\, X = \mathtt{M}, \overline{\mathtt{D}} \rfloor_A = (\lambda x_{A,X}.\{\ell_X = x_{A,X}\} @ \lfloor \overline{\mathtt{D}} \rfloor_A)\, \lfloor \mathtt{M} \rfloor \qquad \lfloor \mathtt{module\ type}\ T = \mathtt{S}, \overline{\mathtt{D}} \rfloor_A = \lfloor \overline{\mathtt{D}} \rfloor_A$$

And that's it. ML modules are an interesting case of programming language research where the type system is quite complex, with involved soundness proofs, while the semantic model is mostly trivial. In the literature, the underlying semantic is sometimes not even mentioned.

Chapter 4

M^ω

In this chapter, we present M^ω (pronounced "Mo-me-ga"), a type system for ML modules that produces inferred signatures in an ML-like syntax extended with type binders $(\exists, \forall, \lambda)$ of F^ω, the higher-order polymorphic lambda calculus. The goal of this system is to provide a specification and an intuition for the core mechanisms informally presented in the previous chapter. The system is proven sound by a full elaboration in F^ω.

Contributions M^ω is strongly inspired by previous works, notably Rossberg et al. [2014] and Russo [2004]. The core contributions of this chapter are the introduction of *transparent existential types* for the soundness proof of the skolemization operation, the *anchoring algorithm* for reconstruction of source signatures from M^ω signatures and the treatment of *abstract signatures* via predicative kind polymorphism.

Code inserts To distinguish M^ω signatures from sources ones, we use the following convention: M^ω signatures have a red background (on the right-hand side), as in:

```
1 module M : sig type t = int end
```

```
1 module M : sig type t = int end
```

Overview We present the type system in Section 4.1: the elaboration of ML signatures into M^ω signatures is presented in Section 4.1.2, along with the key mechanisms of *extrusion* and *skolemization*. Subtyping is presented in Section 4.1.3. The typing of module expressions is detailed in Section 4.1.4.

In Section 4.2, we explain how the right granularity of applicativity can be piggy-backed on the type system by introducing extra *identity* abstract-type fields.

In Section 4.3, we focus on the reconstruction of source signatures from M^ω signatures, a process call *anchoring* that acts as the inverse of the signature elaboration. We first explicit the expressiveness gaps of the source signature syntax, then we present an algorithm that implements anchoring. Finally, we state and prove the properties of this algorithm.

In Section 4.4, we prove the soundness of M^ω via a full elaboration into F^ω. This proof relies on the new mechanism of *transparent existential types* (Section 4.4.4), a weaker form of existential types that supports skolemization. We show that extending F^ω with existential types can be done *internally*, i.e., we define transparent existential types as a library of F^ω (Section 4.4.5). Finally, the actual elaboration is given and proven sound with respect to F^ω typing rules (Sections 4.4.6 and 4.4.7).

M^ω Kinds

$\kappa ::= \star \mid \kappa \to \kappa$

M^ω Types

$\tau ::= \alpha \mid \tau(\overline{\tau}) \mid \lambda\alpha.\tau \mid \ldots$

Environment

$\Gamma ::= \varnothing$ (Empty)
$\mid \Gamma, \alpha$ (Abstract type)
$\mid \Gamma, (Y : \mathcal{C})$ (Functor parameter)
$\mid \Gamma, (A.\mathcal{D})$ (Declaration)

Opacity

$\diamondsuit ::= \triangledown$ (Transparent) $\mid \blacktriangledown$ (Opaque)

M^ω signature

$\mathcal{C} ::= \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}$ (Structural signature)
$\mid \forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}$ (Applicative functor)
$\mid () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ (Generative functor)

M^ω declaration

$\mathcal{D} ::= \mathsf{val}\ x : \tau$ (Values)
$\mid \mathsf{type}\ t = \tau$ (Types)
$\mid \mathsf{module}\ X : \mathcal{C}$ (Modules)
$\mid \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}$ (Module-types)

Figure 7: Syntax of M^ω signatures and typing environment.

4.1 The M^ω type system

We start with an introductory example.

Example 4.1.1. *On the left-hand side, we give a snippet of code, where an ascription is used at several places to introduce abstract types. On the right-hand side, the corresponding M^ω signature is given.*

```
1
2  module type T = sig_C type t = C.t end
3  module X =
4    (struct_A type t = int end : A.T)
5  module X' = struct
6    module X'' =
7      (struct_A type t = bool end : A.T)
8    end
9  module F = (Y : A.T) →
10   (struct_A type t = Y.t × Y.t end : A.T)
11 module G = () →
12   (struct_A type t = int ref end : A.T)
```

```
1  ∃^▽ α1, α2, φ .
2   module type T = λγ.sig type t = γ end
3   module X :
4     sig type t = α1 end
5   module X' : sig
6     module X'' :
7       sig type t = α2 end
8   end
9   module F : ∀β.sig type t = β end →
10    sig type t = φ(β) end
11  module G : () →
12    ∃^▼ α. sig type t = α end
```

The main mechanisms of M^ω typing are at play in the above example. Abstract types introduced by ascriptions on the left-hand side are represented by quantified abstract types in M^ω. While they might be introduced deep in the signature, as in the submodule X'', the quantifier are *extruded* to the top of the signature, making the type variables α_1 and α_2 accessible in all the following declarations. Extrusion out of applicative functors leads to *skolemization*: the type variable φ becomes higher-order. By contrast, type variables are not extruded out of generative functors (line 12). Finally, module type definitions introduce lambda-bound variables.

4.1.1 Overview and technical details

In Figure 7, we introduce the syntax for M^ω-signatures $\mathcal{C}$ and M^ω-declarations $\mathcal{D}$ and typing environments Γ. By convention, we use curvy capitals ($\mathcal{C}$, $\mathcal{D}$, ...) for M^ω-objects.

Types The grammar of M^ω types is the same as the grammar of source types, except that qualified types $Q.t$ are replaced by abstract types α or applied abstract types $\tau(\overline{\tau})$, where α ranges over a collection of abstract-type variables that are distinct from the type variables of the source type language. We use the base kind $\star$ for the kind of first-order types, and $\kappa \to \kappa$ for the higher-order kinds. In the examples, we sometimes use φ instead of α to denote a higher-order type variable. The language is explicitly kinded, as is F^ω. However, we leave kinds mostly implicit in this section for the sake of readability – we display them for the soundness proof in Section 4.4.

Quantifiers positions M^ω-signatures $\mathcal{C}$ use F^ω binders: the universal binder $\forall$ for functor parameters, the existential binder $\exists$ for the body of generative functors and the (type level) lambda binder λ for module-types. We annotate existential quantifiers with an opacity flag $\diamondsuit$ to indicate generativity (using the opaque flag $\blacktriangledown$) or applicativity (using the transparent flag $\triangledown$). This notion is unrelated to transparent signatures. Transparent existentials $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}$ do not appear directly in the grammar of Figure 7 but in the typing judgment for module expressions, which uses existentially quantified signatures of either form $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}$ or $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$. Module-types $\lambda\overline{\alpha}.\mathcal{C}$ are parametric in each type variable α. A parametric signature $\lambda\overline{\alpha}.\mathcal{C}$ may later be transformed into a universal signature $\forall\overline{\alpha}.\mathcal{C}$, an existential signature $\exists^{\diamondsuit}\overline{\alpha}.\mathcal{C}$, or just a normal signature $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. Crucially, module declarations $\mathsf{module}\ X : \mathcal{C}$ do not allow quantifiers in front of $\mathcal{C}$: they must have been extruded away at the top of the signature. Similarly, there is no quantifier inside the codomain of applicative functors $\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$: they must have been skolemized away.

Type equivalence We consider M^ω types up to $\alpha\beta$-equivalence. α-equivalence is standard: bound type variables and functor parameters can be freely renamed, but other identifiers cannot. β-equivalence is defined as the symmetric, reflexive, and transitive and congruent closure of β-reduction, which is defined by the single following rule for type applications:

$$(\lambda\alpha.\sigma)\tau \rightsquigarrow^{\beta} \sigma[\alpha \mapsto \tau]$$

Environments and wellformedness Typing environments contain three types of bindings: an abstract type variable $(\alpha : \kappa)$, a functor argument $Y : \mathcal{C}$, or a declaration $A.\mathcal{D}$. All bindings in Γ are unique (there is no shadowing). Technically, this is achieved by defining mutually recursive *wellformedness* predicates over environments $\vdash \Gamma$, signatures $\Gamma \vdash \mathcal{C}$, and declarations $\Gamma \vdash \mathcal{D}$, along with a *well-kindness* predicate for types $\Gamma \vdash \tau : \kappa$. The rules are standard and given in Figure 8. We write $\cdot \notin \Gamma$ as a shortcut for $\cdot \notin \mathsf{dom}(\Gamma)$. The only subtlety comes from the wellformedness of structural signatures:

$$\frac{\mathsf{disjoint}(\overline{\mathcal{D}}) \qquad \Gamma \vdash \overline{\mathcal{D}}}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

Here, we use a helper predicate $\mathsf{disjoint}(\overline{\mathcal{D}})$ to ensure that all field identifiers present in the list $\overline{\mathcal{D}}$ are pair-wise distinct.

Wellformedness conditions As a simplifying convention for the rest of this chapter, we consider wellformedness of the environment as a precondition to all rules. Alternatively, we could have added wellformedness preconditions sparingly, only to the rules that are *leaves* of the derivation, i.e., that do not have another typing or subtyping as a premise. Then, we would have shown that typing and subtyping always implies the wellformedness of the environment: a rule is either a leaf and has an explicit wellformedness premise, or is not a leaf and the wellformedness of the environment is implied by another premise by induction hypothesis. Here, we do not see any benefit in this latter approach and we use the simplifying convention of having wellformedness everywhere.

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \qquad \alpha \notin \Gamma}{\vdash \Gamma, (\alpha : \kappa)} \qquad \frac{Y \notin \Gamma \qquad \Gamma \vdash \mathcal{C}}{\vdash \Gamma, Y : \mathcal{C}} \qquad \frac{\Gamma \vdash \tau : \star \qquad A.x \notin \Gamma}{\vdash \Gamma, A.\mathsf{val}\ x : \tau} \qquad \frac{\Gamma \vdash \tau : \kappa \qquad A.t \notin \Gamma}{\vdash \Gamma, A.\mathsf{type}\ t = \tau}$$

$$\frac{\Gamma \vdash \mathcal{C} \qquad A.X \notin \Gamma}{\vdash \Gamma, A.\mathsf{module}\ X : \mathcal{C}} \qquad \frac{\overline{\alpha} \notin \Gamma \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C} \qquad A.T \notin \Gamma}{\vdash \Gamma, A.\mathsf{module\ type}\ T = \lambda(\overline{\alpha : \kappa}).\mathcal{C}}$$

(a) Wellformedness of environments

$$\frac{\mathsf{disjoint}(\overline{\mathcal{D}}) \qquad \Gamma \vdash \overline{\mathcal{D}}}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \frac{\Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}_a \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash \forall \overline{(\alpha : \kappa)}.\mathcal{C}_a \to \mathcal{C}} \qquad \frac{\Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash () \to \exists^{\blacktriangledown} \overline{(\alpha : \kappa)}.\mathcal{C}}$$

(b) Wellformedness of signatures

$$\frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \mathsf{val}\ x : \tau} \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathsf{type}\ t = \tau} \qquad \frac{\Gamma \vdash \mathcal{C}}{\Gamma \vdash \mathsf{module}\ X : \mathcal{C}} \qquad \frac{\Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash \mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}}$$

(c) Wellkindness of declarations

$$\frac{\vdash \Gamma \qquad (\alpha : \kappa) \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \kappa \to \kappa' \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash (\tau\ \sigma) : \kappa'} \qquad \frac{\Gamma, (\alpha : \kappa) \vdash \tau : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).\tau : \kappa \to \kappa'}$$

(d) Wellkindness of types

Figure 8: Wellformedness rules for environments, signatures, declarations, and types.

Judgments

- $\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit} \overline{\alpha}.\mathcal{C}$ and $\Gamma \vdash \mathtt{D} : \exists^{\diamondsuit} \overline{\alpha}.\mathcal{D}$ – typechecking of modules and bindings, defined in Figure 12 and discussed in Section 4.1.4.
- $\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$ and $\Gamma \vdash \mathtt{D} : \lambda \overline{\alpha}.\mathcal{C}$ – typechecking of signatures and declarations, defined in Figure 9 and discussed in Section 4.1.2.
- $\Gamma \vdash \mathtt{u} : \tau$ and $\Gamma \vdash \mathtt{e} : \tau$ – elaboration of core-language types and type-checking of core-language expressions.
- $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ and $\Gamma \vdash \mathcal{D} < \mathcal{D}'$ – subtyping of signatures and declarations, defined in Figure 11 and discussed in Section 4.1.3.

4.1.2 Signatures type-checking

The key concepts of M^ω can be illustrated with the typechecking of signatures $\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$ (and declarations), which translates a source signature $\mathtt{S}$ into its M^ω counterpart $\lambda \overline{\alpha}.\mathcal{C}$, making the set of abstract type $\overline{\alpha}$ explicit. The typechecking of signatures also acts as an *elaboration*: source signatures are checked and translated into M^ω-signatures. We refer to both *typechecking* and *elaboration*, using the former to emphasize the wellformedness check and the latter to emphasize the translation into the M^ω-signature language. The full set of rules is given in Figure 9 and discussed below.

$$\textsc{M-Typ-Sig-ModType}\ \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \vdash P.T : \lambda\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Sig-LocalModType}\ \frac{A.\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \in \Gamma}{\Gamma \vdash A.T : \lambda\overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Sig-GenFct}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Sig-AppFct}\ \frac{\Gamma \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

$$\textsc{M-Typ-Sig-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : \lambda\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \textsc{M-Typ-Sig-Trans}\ \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

(a) Signature typing rules.

$$\textsc{M-Typ-Decl-Val}\ \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{val}\ x : \mathtt{u}) : (\mathsf{val}\ x : \tau)} \qquad \textsc{M-Typ-Decl-Type}\ \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\ t = \mathtt{u}) : (\mathsf{type}\ t = \tau)} \qquad \textsc{M-Typ-Decl-TypeAbs}\ \Gamma \vdash_A (\mathtt{type}\ t = A.t) : \lambda\alpha.(\mathsf{type}\ t = \alpha)$$

$$\textsc{M-Typ-Decl-Mod}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\ X : \mathtt{S}) : \lambda\overline{\alpha}.(\mathsf{module}\ X : \mathcal{C})} \qquad \textsc{M-Typ-Decl-ModType}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ \ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

$$\textsc{M-Typ-Decl-Empty}\ \Gamma \vdash_A \varnothing : \varnothing \qquad \textsc{M-Typ-Decl-Seq}\ \frac{\Gamma \vdash_A \mathtt{D}_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}}}$$

(b) Declaration typing rules.

$$\textsc{M-Typ-Type-Path}\ \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{type}\ t = \tau \in \overline{\mathcal{D}}}{\Gamma \vdash P.t : \tau} \qquad \textsc{M-Typ-Type-Local}\ \frac{A.\mathsf{type}\ t = \tau \in \Gamma}{\Gamma \vdash A.t : \tau}$$

(c) Extension to the core-language typing rules

Figure 9: Signature and declaration typing rules.

Extrusion

During the typechecking of declarations, an abstract-type variable is introduced for each abstract type field and its binder is *extruded* step by step until a scope barrier is reached. Doing so, the scope of the binder extends to the whole region where the abstract type field is accessible by a path. This mechanism can be seen in action in Figure 10 An abstract type declaration introduces an abstract type variable α that is λ-bound:

$$\frac{}{\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \lambda\alpha.(\mathsf{type}\, t = \alpha)}\ \textsc{M-Typ-Decl-TypeAbs}$$

Since the name t is also accessible in the following declarations, the λ-binder for α must be *extruded* (we also use the term *lifted*) to enclose the whole region where t, hence α, is accessible. This lifting is performed in two places:

$$\frac{\Gamma \vdash_A \mathtt{D}_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}}}\ \textsc{M-Typ-Decl-Seq}$$

$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S}) : \lambda\overline{\alpha}.(\mathsf{module}\, X : \mathcal{C})}\ \textsc{M-Typ-Decl-Mod}$$

First, when merging a list of declarations in Rule M-Typ-Decl-Seq: the two sets of abstract types $\overline{\alpha}$ and $\overline{\alpha}_1$ are merged together in front of the list of declarations. Second, lifting also occurs when typing a module declaration (Rule M-Typ-Decl-Mod) where the set of abstract types $\overline{\alpha}$ introduced in the elaboration of $\mathtt{S}$ are lifted to the outside of the declaration. In a nutshell, abstract types introduced by submodules are not bound at the declaration of the submodule, but lifted to the enclosing signature.

By contrast, when typing module-type declarations (Rule M-Typ-Decl-ModType), the binder is not lifted, as the module-type declaration defines its own scope. The declarations inside $\mathcal{C}$ are not accessible in the rest of the enclosing signature.:

$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\, T = \lambda\overline{\alpha}.\mathcal{C})}\ \textsc{M-Typ-Decl-ModType}$$

Typing declarations of value or type fields relies on the elaboration judgment $\Gamma \vdash \mathtt{u} : \tau$ of core-language types into F^ω which is left abstract:

$$\frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{val}\, x : \mathtt{u}) : (\mathsf{val}\, x : \tau)}\ \textsc{M-Typ-Decl-Val} \qquad \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\, t = \tau)}\ \textsc{M-Typ-Decl-Type}$$

Still, we extend the core-language judgment with rules to translate qualified types into M^ω types:

$$\frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{type}\ t = \tau \in \overline{\mathcal{D}}}{\Gamma \vdash P.t : \tau}\ \textsc{M-Typ-Type-Path} \qquad \frac{A.(t :\ \mathsf{type}\ \tau) \in \Gamma}{\Gamma \vdash A.t : \tau}\ \textsc{M-Typ-Type-Local}$$

Signatures

Structural signatures The typing of declarations is injected in the typing of signatures by the rule for structural signatures:

$$\frac{\Gamma \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : \lambda\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}\ \textsc{M-Typ-Sig-Str}$$

This is the last place where extrusion happens (see M-Typ-Decl-Type, M-Typ-Decl-Mod, M-Typ-Decl-Seq for the other rules): the abstract types $\overline{\alpha}$ introduced by the declarations are lifted in front of the whole signature.

```
sig_A
  type t = A.t
  module type T =
    sig_C type t = C.t end
  module X : A.T
  module G : () → A.T
  module F : (Y : A.T) → A.T

  type v = A.t × A.X.t
  type w = A.F(A.X).t
end
```

```
λα1, α2, φ.sig
  type t = α1
  module type T =
    λα.sig type t = α end
  module X : sig type t = α2 end
  module G : () → ∃▼α. sig type t = α end
  module F : ∀β.sig type t = β end →
       sig type t = φ(β) end
  type v = α1 × α2
  type w = φ(α2)
end
```

Figure 10: On the left-hand side, we consider a source signature. The corresponding signature resulting from typechecking in M^ω is given on the right-hand side. Here, the signature is mostly composed of type declarations because it displays the interesting mechanisms of the type system, but in real use cases, signatures are mostly composed of value declarations. Both the abstract type variables α_1, α_2, and φ are extruded at the top-level of the signature, even though they come from different levels of the signature. The intermediate definition of the module-type $\mathtt{T}$ is inlined. The applicative functor introduces a higher-order type φ, while the generative one introduces only a base kind type variable that is not extruded.

Module-types Module-type definitions are inlined by the following two rules:

$$\textsc{M-Typ-Sig-ModType}\quad \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \vdash P.T : \lambda\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Sig-LocalModType}\quad \frac{A.(T :\ \mathsf{module\ type}\ \lambda\overline{\alpha}.\mathcal{C}) \in \Gamma}{\Gamma \vdash A.T : \lambda\overline{\alpha}.\mathcal{C}}$$

Therefore, M^ω signatures do not have a counterpart for module-type paths $Q.T$ that occur in source signatures.

Transparent signatures Rule M-Typ-Sig-Trans displays an interesting mechanism:

$$\textsc{M-Typ-Sig-Trans}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

The source signature $\mathtt{S}$ is first elaborated into an M^ω-signature $\lambda\overline{\alpha}.\mathcal{C}'$. The M^ω-signature $\mathcal{C}$ of the path P is then obtained by module typing. From there, we use subtyping to compare the two signatures. But there is a catch: $\lambda\overline{\alpha}.\mathcal{C}$ is parametric, whereas $\mathcal{C}'$ is not. Therefore, we need to first find an *instantiation* of the abstract types $\overline{\alpha}$ by some concrete types $\overline{\tau}$. Finding such instantiation is non-trivial, especially in the presence of higher-order types. This problem is discussed in more details along with subtyping in Section 4.1.3. Once the instantiation has been found, the instantiated signature $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$ can be compared against $\mathcal{C}'$. The result of the elaboration is the instantiation $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. Notably, no new abstract type is introduced. This follows the intuition that transparent signatures share all their type fields with another pre-existing module, and therefore all abstract type variables have always already been introduced.

Generative functors Here, the abstract types $\overline{\alpha}$ introduced by the codomain of the functor are not extruded in front of the whole signature. Instead, they are left as an

$$\textsc{M-Typ-Sig-GenFct}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Sub-Sig-Struct}\quad \frac{\overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}}_0 < \overline{\mathcal{D}}'}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} < \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end}}$$

$$\textsc{M-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}]}{\Gamma \vdash () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} < () \to \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}'}$$

$$\textsc{M-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] < \mathcal{C}'}{\Gamma \vdash \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} < \forall \overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}'}$$

(a) Subtyping rules for signatures

$$\textsc{M-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) < (\mathsf{val}\ x : \tau)$$

$$\textsc{M-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) < (\mathsf{type}\ t = \tau)$$

$$\textsc{M-Sub-Decl-Mod}\quad \frac{\Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) < (\mathsf{module}\ X : \mathcal{C}')}$$

$$\textsc{M-Sub-Decl-ModType}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}' \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' < \mathcal{C}}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}) < (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}')}$$

(b) Subtyping rules for declarations. Subtyping between lists of declaration is just a map.

Figure 11: M^ω– Subtyping rules (signatures and declarations)

opaque existential signature $\exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}$ for the functor codomain. Indeed, every instantiation of the functor should generate *new* (incompatible) abstract types $\overline{\alpha}$, as required for generativity.

Applicative functors By contrast, applicative functors should not be assigned a signature of the form $\forall \overline{\alpha}.\mathcal{C} \to \exists^{\blacktriangledown} \overline{\beta}.\mathcal{C}'$ where all applications would produce new abstract types, nor $\lambda \beta.\forall \overline{\alpha}.\mathcal{C} \to \mathcal{C}'$ where all applications would share the same types regardless of their argument. Instead, the following rule follows the solution of Biswas [1995] and reused in Russo [2004] and *F-ing* (Rossberg et al. [2014]):

$$\textsc{M-Typ-Sig-AppFct}\quad \frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

That is, we use higher-order abstract types $\overline{\beta'}$ and apply each β' to the universally quantified variables $\overline{\alpha}$ to capture the fact that each abstract type β' is *some type function* of the arguments. This gives a signature of the form $\lambda \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C} \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]$. This extrusion out of an arrow type and out of a universal binder is called *skolemization* and was first introduced by Biswas [1995] for higher-order functors. Here, it is merely a manipulation of types and signatures that does not pose a challenge for type soundness. In a sense, we can manipulate types in any way as long as they remain wellformed. One key technical aspect of M^ω is to justify a similar skolemization but for the type of module expressions, as described in Section 4.4.

Here and until Section 4.2, M^ω applicative functor have a type-level applicativity, like Moscow ML: the abstract types $\overline{\beta'}$ of the codomain depend on the abstract types $\overline{\alpha}$ of the parameter, not on the whole module. We explain how to change it to module-level applicativity as in OCaml in Section 4.2.

4.1.3 Subtyping

The subtyping judgment $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ and its helper judgment $\Gamma \vdash \mathcal{D} < \mathcal{D}'$ check that a signature $\mathcal{C}$ is *more restrictive* than a signature $\mathcal{C}'$. In practice, this boils down to $\mathcal{C}$ having more

fields and introducing fewer abstract types than $\mathcal{C}'$. Intuitively, it can be understood as "*there are less modules with the signature $\mathcal{C}$ than with the signature $\mathcal{C}'$* ". Overall, subtyping in M^ω is a combination of three orthogonal mechanisms:

- structural subtyping between record types, including deletion and reordering of fields.
- subtyping between function types: covariant on their codomains and contravariant on their domains.
- subtyping between universal types and between existential types via instantiation. Originally introduced as the (sub) rule for "type containment" in Mitchell [1988], or sometimes called *subtyping à la F^η*, it allows partial instantiation of polymorphic types.

The full set of subtyping rules is given in Figure 11 and discussed below.

Structural signatures The key rule is the comparison of two structural signatures:

$$\textsc{M-Sub-Sig-Struct}\quad \frac{\overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}}_0 < \overline{\mathcal{D}}'}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} < \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end}}$$

A subset $\overline{\mathcal{D}}_0$ of the fields $\overline{\mathcal{D}}$ of the left-hand side signature is matched and subtyped against the full set of fields of the right-hand side signature $\overline{\mathcal{D}}'$. This subset $\overline{\mathcal{D}}_0$ can be reordered and does not have to contain all the fields of the right-hand side signature. Algorithmically, it is easy to find $\overline{\mathcal{D}}_0$ from the combination of $\overline{\mathcal{D}}$ and $\overline{\mathcal{D}}'$: for each declaration of $\overline{\mathcal{D}}'$, we must find a corresponding declaration in $\overline{\mathcal{D}}$ with the same identifier, and there exists at most one, since identifiers are pairwise distinct in a structural signature.

Declarations Subtyping between declarations boils down to a matching of definitions, as we assume no subtyping on the core language. The identifiers always have to be the same on both sides. For values and type declarations, we have:

$$\textsc{M-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) < (\mathsf{val}\ x : \tau) \qquad \textsc{M-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) < (\mathsf{type}\ t = \tau)$$

Subtyping propagates down to submodules, which gives the following rule:

$$\textsc{M-Sub-Decl-Mod}\quad \frac{\Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) < (\mathsf{module}\ X : \mathcal{C}')}$$

There are several possible choices for the subtyping rule between two module type fields. In *F-ing* (Rossberg et al. [2014]), they have a relaxed rule where they require subtyping in both directions between the two definitions: it only allows for a *different ordering* of the same set of fields between the two definitions. We could also make the rule more restricted, requiring the same signature (up to M^ω signature equivalence defined in Section 4.1.1) on both sides. OCaml uses a criterion that is in-between, requiring code-free subtyping between the two declarations. It makes sense when the names of the fields are kept in the resulting signature: changing the definition of $\mathtt{type}\, t = \mathtt{u}$ when there are occurrences of t remaining in the signature should only be allowed if the change of definition is code-free. This is discussed for ZipML in Section 5.3.5. In M^ω, definitions are inlined, so changing the definition does not affect the rest of the signature. Yet, for coherence we reuse the same criterion.

$$\textsc{M-Sub-Decl-ModType}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}' \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' < \mathcal{C} \qquad \lfloor \mathcal{C} \rfloor = \lfloor \mathcal{C}' \rfloor}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}) < (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}')}$$

Where define the *dynamic part* of $\mathcal{C}$, written $\lfloor \mathcal{C} \rfloor$, as a pseudo-untyped signature:

$$\begin{array}{rcl} \lfloor () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} \rfloor & \triangleq & () \to \mathcal{C} \\ \lfloor \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \rfloor & \triangleq & \lfloor \mathcal{C}_a \rfloor \to \lfloor \mathcal{C} \rfloor \\ \lfloor \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rfloor & \triangleq & \mathsf{sig}\ \lfloor \overline{\mathcal{D}} \rfloor\ \mathsf{end} \end{array} \qquad \begin{array}{rcl} \lfloor \mathtt{val}\ x : \_ \rfloor & \triangleq & \mathsf{val}\ x \\ \lfloor \mathtt{module}\ X : \mathcal{C} \rfloor & \triangleq & \mathtt{module}\ X : \lfloor \mathcal{C} \rfloor \\ \lfloor \mathtt{type}\ t = \_ \rfloor & \triangleq & \varnothing \\ \lfloor \mathtt{module\ type}\ T = \_ \rfloor & \triangleq & \varnothing \end{array}$$

The equality between dynamic parts is purely syntactic.

Generative functors The rule for generative functors M-Sub-Sig-GenFct shows the mechanism of subtyping by *instantiation* of quantifiers:

$$\textsc{M-Sub-Sig-GenFct}\ \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}]}{\Gamma \vdash () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} < () \to \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}'}$$

It can be read as a two-step process, where we first check the subtyping between $\exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}$ and $\exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}'$, which in turn amounts to finding an instantiation of $\overline{\alpha}'$ by some type $\overline{\tau}'$ (that might use the $\overline{\alpha}$). While this is one of the standard ways of specifying subtyping for existential types, as done in Mitchell [1988]; Russo [2004]; Rossberg et al. [2014], it quite permissive, as it does not require both sets of variables to be the same. However, it is algorithmically challenging in the presence of higher-order abstract types, and could potentially lead to undecidability of subtyping.

This problem has already been identified in the literature Biswas [1995]; Russo [2004]; Rossberg et al. [2014]. We reuse the argument with the terminology of *F-ing* (Rossberg et al. [2014]), which we sum-up below. Decidability follows from the fact that subtyping $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ is actually only checked when the right-hand-side signature $\mathcal{C}'$ has an additional property that makes the instantiation easy to compute.

More precisely, they define:

- *rooted type variables*: Let us first consider a type variable α of kind $\star$. We say that α is rooted in a signature $\mathcal{C}'$ if the signature contains a type field of the form $\mathsf{type}\ t = \alpha$ in a strictly positive position. The key observation is that rooted type variables can be easily instantiated during subtyping. Indeed, for the subtyping to succeed, there have to be some corresponding type field of the form $\mathsf{type}\ t = \tau$ with the same identifier t occurring in a strictly positive part of $\mathcal{C}$. From the non-variance of declaration subtyping, we know that the only possible instantiation of α has to be τ. Therefore, the instantiation can be found for rooted type variables, using the two signatures $\mathcal{C}$ and $\mathcal{C}'$.
  This argument extends to higher-order types. For functors with one argument, a type variable φ is rooted in $\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$ if $\mathcal{C}$ contains a type field of the form $\mathsf{type}\ t = (\varphi\, \overline{\alpha})$ in a strictly positive position. Again, the left-hand-side signature $\mathcal{C}$ must have a corresponding type field of the form $\mathsf{type}\ t = \tau$. Therefore, the only possible instantiation of φ is $\lambda \overline{\alpha}.\tau$
  For several arguments, the set of type variables $\overline{\alpha}$ has to be the concatenation of all universally quantified variables from the different levels of functors.
- *explicit signatures* are signatures where all quantified abstract types are rooted. Signatures obtained by elaboration from source signatures are always explicit signatures.
- *valid signatures* are signatures where all functors have an explicit signature on the left-hand side of the arrow.

From there, they show that typechecking only produces valid signatures, and that subtyping is only checked with an explicit signature on the right-hand side and a valid signature on the left-hand side.

An interesting consequence is that decidability of subtyping (and therefore, decidability of typechecking) crucially relies on the fact that the elaboration of source signatures always

produces explicit signatures. If we where to allow the user to input non-explicit signatures – either with an unrestricted **`module type of`** construct, or directly in M^ω-signatures syntax – we would lose decidability. This hints at the fact that ML-modules are not just merely a *mode of use of* F^ω (as written in *F-ing* (Rossberg et al. [2014])), but also a *sweet spot*, where changes seen as minor from M^ω could easily break decidability.

Applicative functors Subtyping between functors combines both instantiation and subtyping between arrow types:

$$\text{M-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] < \mathcal{C}'}{\Gamma \vdash \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} < \forall \overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}'}$$

As usual, subtyping is contravariant for the argument of applicative functors. The same instantiation $[\overline{\alpha} \mapsto \overline{\tau}]$ is used for both the domain and codomain.

4.1.4 Module Expressions type-checking

Typechecking of expressions $\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit} \overline{\alpha}.\mathcal{C}$ infers an M^ω-signature $\exists^{\diamondsuit} \overline{\alpha}.\mathcal{C}$ given a source module `M`. Technically, the judgment should be read $\Gamma \vdash^{\diamondsuit} \mathtt{M} : \exists^{\diamondsuit} \overline{\alpha}.\mathcal{C}$ where the opacity flag on the judgment is a typing mode that is the same as the opacity mode of the existential quantifier. As we use opacity flags as modes, we also refer to the opaque flag $\blacktriangledown$ as the *generative* mode, and to the transparent flag $\triangledown$ as the *applicative* mode.

To lighten the notation, we usually omit the mode on the judgment except when it is generative and there is no existential type to enforce it. Thus, when we write $\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown} \varnothing.\mathcal{C}$ or $\Gamma \vdash \mathtt{M} : \exists^{\triangledown} \varnothing.\mathcal{C}$ when $\overline{\alpha}$ is empty, we actually mean $\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathcal{C}$ and $\Gamma \vdash^{\triangledown} \mathtt{M} : \mathcal{C}$. The same convention applies to typing rules for bindings M-Typ-Decl-*. Typing rules for module expressions and bindings are given in Figure 12.

Skolemization The rule for applicative functors is a crucial one:

$$\text{M-Typ-Mod-AppFct}\quad \frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathtt{M} : \exists^{\triangledown} \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^{\triangledown} \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\alpha)}\right]}$$

In order to share the abstract types $\overline{\beta}$ produced by the inference of the body of the functor `M`, we *skolemize* them out of the universal quantification (and out of the arrow type) by making them higher-order, and applying them to the universally quantified variables $\overline{\alpha}$. However, this is unsound when the abstract types are opaque existential types. We thus enforce transparent existential types for the body of the functor. The technical reasons behind this restriction are detailed in Section 4.4. In a nutshell, skolemization of transparent existential types is always sound, and applicative functor only need this restricted form of existentials.

One might wonder why the corresponding rule M-Typ-Sig-AppFct for typing signatures of applicative functors does not feature a similar restriction. The key difference is that typing of signatures is just a mere manipulation of types where everything is possible, as long as the resulting signatures are wellformed. By contrast, for the typing of module expressions, the soundness relies on the ability to actually produce a term of the corresponding type in F^ω.

Propagation of modes Signatures with transparent existentials are inferred by default and are *required* for the body of applicative functors, as seen above. Module expressions that are inherently generative, such as calling generative functors or computing impure core

expressions (or unpacking first-class modules), can only be typed with opaque existential signatures, in generative mode:

$$\textsc{M-Typ-Mod-GenFct}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Mod-AppGen}\ \frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

Modules can be *downgraded* from applicative to generative via subsumption, but not the other way around:

$$\textsc{M-Typ-Mod-Seal}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

All other rules are agnostic of the typing mode. With the convention that M-Typ-Mod-Seal is only used when the generative mode is required for the premise of another rule, i.e., applicative signatures are inferred by default, the system is syntax directed.

Structures are regular, i.e., all bindings have the same mode. This is enforced by the Rule M-Typ-Bind-Seq for sequences, where the mode must be the same on both premises:

$$\textsc{M-Typ-Mod-Struct}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\diamond}\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \textsc{M-Typ-Bind-Seq}\ \frac{\Gamma \vdash_A \mathtt{B} : \exists^{\diamond}\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

Bindings The rules for bindings assume a given core-language expression typing judgment $\Gamma \vdash^{\diamond} \mathtt{e} : \tau$ equipped with a mode that tracks the presence of effects. As explained in Section 2.2.2, it is not present in current OCaml, where it is the user's responsibility to use generative functors in such cases:

$$\textsc{M-Typ-Bind-Let}\ \frac{\Gamma \vdash^{\diamond} \mathtt{e} : \tau}{\Gamma \vdash_A{}^{\diamond} (\mathtt{let}\ x = \mathtt{e}) : (\mathsf{val}\ x : \tau)} \qquad \textsc{M-Typ-Bind-Type}\ \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A{}^{\diamond} (\mathtt{type}\ t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

$$\textsc{M-Typ-Bind-Mod}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\ X = \mathtt{M}) : (\exists^{\diamond}\overline{\alpha}.\mathsf{module}\ X : \mathcal{C})} \qquad \textsc{M-Typ-Bind-ModType}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

The core-language expression typing is extended by the following rules for qualified variables:

$$\textsc{M-Typ-Type-Path}\ \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{val}\ x : \tau \in \overline{\mathcal{D}}}{\Gamma \vdash^{\diamond} P.x : \tau} \qquad \textsc{M-Typ-Type-Local}\ \frac{A.\mathsf{val}\ x : \tau \in \Gamma}{\Gamma \vdash^{\diamond} A.x : \tau}$$

Paths The typing of paths, which may contain applications of applicative functors, is defined by the following rules:

$$\textsc{M-Typ-Mod-Var}\ \frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \vdash Y : \mathcal{C}} \qquad \textsc{M-Typ-Mod-Local}\ \frac{(A.X : \ \mathsf{module}\ \mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C}}$$

$$\textsc{M-Typ-Mod-AppApp}\ \frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

There is an implicit subtyping check at functor application.

Introduction of abstract types Ascription to a signature $\mathtt{S}$ that elaborates to a parametric signature $\lambda\overline{\alpha}.\mathcal{C}$ returns a signature $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}$ with transparent abstract types:

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}'} \text{ M-Typ-Mod-Ascr}$$

This rule has some resemblance with Rule M-Typ-Sig-Trans for typechecking a transparent signature ($= P < \mathtt{S}$): in both cases, we check that the M^{ω} signature of P is a subtype of the M^{ω}-signature $\lambda\overline{\alpha}.\mathcal{C}'$ of $\mathtt{S}$. By contrast, however, we here drop the matching substitution in the result signature $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}'$ and instead introduce the abstract types $\overline{\alpha}$. If the signature is not parametric, i.e., does not bind abstract types, the result does not introduce new abstract types either. In particular, transparent ascription ($P <: \mathtt{S}$), which is syntactic sugar for ($P : (= P < \mathtt{S})$), i.e., the opaque ascription of P to the transparent signature ($= P < \mathtt{S}$), behaves as expected, filtering out components of P as prescribed by $\mathtt{S}$ but without creating new abstract types. Note that applications of an applicative functor (Rule M-Typ-Mod-AppApp) do not introduce new abstract types *per se*, but *applications* of *already existing* higher-order abstract types – which is the key to the sharing between different applications of the same (or an equivalent) functor to the same (or equivalent) arguments.

The only other construct that introduces abstract types is abstract-type binding:

$$\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \exists^{\diamondsuit}\alpha.(\mathsf{type}\, t = \alpha) \quad \text{M-Typ-Bind-AbsType}$$

Projection and signature avoidance As explained in Section 2.3, typechecking the projection of a submodule $\mathtt{M}.X$ is often a source of signature avoidance: the dependencies of the source signature of X might become dangling after the other components of the signature of $\mathtt{M}$ have been lost. However, M^{ω} signatures do not have internal dependencies; they are non-dependent records, as all paths present in concrete type definitions have been inlined and binders for abstract types have been lifted. This gives a simple projection rule:

$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit}\overline{\alpha}.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad (\mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}) \subseteq \overline{\alpha}'}{\Gamma \vdash \mathtt{M}.X : \exists^{\diamondsuit}\overline{\alpha}'.\mathcal{C}} \text{ M-Typ-Mod-Proj}$$

In principle, the Rule M-Typ-Mod-Proj could return the signature $\exists^{\diamondsuit}\overline{\alpha}.\mathcal{C}$, leaving all variables in scope after projection. However, it also performs some form of garbage collection by just keeping the subset $\overline{\alpha}'$ of abstract types $\overline{\alpha}$ that appear free in the submodule signature $\mathcal{C}$ so as to avoid keeping useless, unreachable abstract types.

4.2 Identity, Aliasing, and Type Abstraction

So far, our system handles applicativity with a type-level granularity of applicativity, as promoted by Moscow ML Russo [2001] and *F-ing* (Rossberg et al. [2014]). In this section, we present the introduction of identity types in a simple *source-to-source* transformation as a way to piggy-back the OCaml module-level granularity on top of type-level applicativity. We also present a derived type-system that treats module-level applicativity in a primitive way. Finally, we state and prove a property of identity types that hints at abstraction safety (without obtaining it).

4.2.1 A source-to-source transformation

As explained in Section 2.2.2, applicativity of functors relies on a criterion for *module equivalence* that gives rise to three notions of applicativity:

$$\textsc{M-Typ-Mod-Var}\ \frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \vdash Y : \mathcal{C}} \qquad \textsc{M-Typ-Mod-Local}\ \frac{(A.X : \ \mathsf{module}\ \mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C}} \qquad \textsc{M-Typ-Mod-Seal}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\triangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-Struct}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\diamond} \overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\diamond} \overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

$$\textsc{M-Typ-Mod-Ascr}\ \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown} \overline{\alpha}.\mathcal{C}'}$$

$$\textsc{M-Typ-Mod-AppFct}\ \frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathtt{M} : \exists^{\triangledown} \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^{\triangledown} \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]} \qquad \textsc{M-Typ-Mod-GenFct}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-AppApp}\ \frac{\Gamma \vdash P : \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]} \qquad \textsc{M-Typ-Mod-AppGen}\ \frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash P() : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-Proj}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad (\mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}) \subseteq \overline{\alpha}'}{\Gamma \vdash \mathtt{M}.X : \exists^{\diamond} \overline{\alpha}'.\mathcal{C}}$$

(a) Module expression (and path) typing rules

$$\textsc{M-Typ-Bind-Let}\ \frac{\Gamma \vdash^{\diamond} \mathtt{e} : \tau}{\Gamma \vdash_A{}^{\diamond} (\mathtt{let}\ x = \mathtt{e}) : (\mathsf{val}\ x : \tau)} \qquad \textsc{M-Typ-Bind-Type}\ \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A{}^{\diamond} (\mathtt{type}\ t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

$$\textsc{M-Typ-Bind-AbsType}\ \Gamma \vdash_A (\mathtt{type}\ t = A.t) : \exists^{\diamond} \alpha.(\mathsf{type}\ t = \alpha) \qquad \textsc{M-Typ-Bind-Mod}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\ X = \mathtt{M}) : (\exists^{\diamond} \overline{\alpha}.\mathsf{module}\ X : \mathcal{C})}$$

$$\textsc{M-Typ-Bind-ModType}\ \frac{\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ \ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C})} \qquad \textsc{M-Typ-Bind-Empty}\ \Gamma \vdash_A \varnothing : \varnothing$$

$$\textsc{M-Typ-Bind-Seq}\ \frac{\Gamma \vdash_A \mathtt{B} : \exists^{\diamond} \overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\diamond} \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\diamond} \overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

(b) Binding typing rules

$$\textsc{M-Typ-Type-Path}\ \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{val}\ x : \tau \in \overline{\mathcal{D}}}{\Gamma \vdash^{\diamond} P.x : \tau} \qquad \textsc{M-Typ-Type-Local}\ \frac{A.(x : \ \mathsf{val}\ \tau) \in \Gamma}{\Gamma \vdash^{\diamond} A.x : \tau}$$

(c) Extension of core-language expression typing

Figure 12: Module and binding typing rules.

Tagging

$$\mathtt{Tag}\,\{\mathtt{M}\} \triangleq \mathtt{struct}_A\ \mathtt{module}\ \mathit{Val} = \mathtt{M}\ \mathtt{type}\ \mathit{id} = A.\mathit{id}\ \mathtt{end}$$
$$\mathtt{Tag}\,\{\mathtt{S}\} \triangleq \mathtt{sig}_A\ \mathtt{module}\ \mathit{Val} : \mathtt{S}\ \mathtt{type}\ \mathit{id} = A.\mathit{id}\ \mathtt{end}$$

Paths

$$[\![A.X]\!] \triangleq A.X \qquad [\![P.X]\!] \triangleq [\![P]\!].\mathit{Val}.X$$
$$[\![Y]\!] \triangleq Y \qquad [\![P(P')]\!] \triangleq [\![P]\!].\mathit{Val}([\![P']\!])$$

Module expressions

$$[\![\mathtt{M}.X]\!] \triangleq [\![\mathtt{M}]\!].\mathit{Val}.X \quad [\![P()]\!] \triangleq [\![P]\!].\mathit{Val}()$$
$$[\![(P : \mathtt{S})]\!] \triangleq ([\![P]\!] : [\![\mathtt{S}]\!])$$
$$[\![() \to \mathtt{M}]\!] \triangleq \mathtt{Tag}\,\{() \to [\![\mathtt{M}]\!]\}$$
$$[\![(Y : \mathtt{S}) \to \mathtt{M}]\!] \triangleq \mathtt{Tag}\,\{(Y : [\![\mathtt{S}]\!]) \to [\![\mathtt{M}]\!]\}$$
$$[\![\mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}]\!] \triangleq \mathtt{Tag}\,\{\mathtt{struct}_A\ [\![\overline{\mathtt{B}}]\!]\ \mathtt{end}\}$$

Signatures

$$[\![A.T]\!] \triangleq A.T \quad [\![P.T]\!] \triangleq [\![P]\!].\mathit{Val}.T$$
$$[\![(= P < \mathtt{S})]\!] \triangleq (= [\![P]\!] < [\![\mathtt{S}]\!])$$
$$[\![() \to \mathtt{S}]\!] \triangleq \mathtt{Tag}\,\{() \to [\![\mathtt{S}]\!]\}$$
$$[\![(Y : \mathtt{S}_a) \to \mathtt{S}]\!] \triangleq \mathtt{Tag}\,\{(Y : [\![\mathtt{S}_a]\!]) \to [\![\mathtt{S}]\!]\}$$
$$[\![\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}]\!] \triangleq \mathtt{Tag}\,\{\mathtt{sig}_A\ [\![\overline{\mathtt{D}}]\!]\ \mathtt{end}\}$$

Figure 13: Source-to-source transformation introducing identity tags for structures and functors using two reserved identifiers `id` and `Val`. Bindings and declarations and types are transformed by immediate map over submodules and submodule-types.

- *type-level applicativity*: the two modules have the same type fields
- *value-level applicativity*: the two modules have the same type and value fields
- *module-level applicativity*: the two modules are statically known as being equal

To obtain the *abstraction safety* provided by the last two options, Rossberg et al. [2014] introduced *semantic paths*: tagging value and module fields with *phantom* abstract types and using the type sharing mechanism to track value or module sharing. Then, type-level applicativity can be transformed into either value level or module-level (*à la* OCaml) by marking either all values or only modules.

However, as phantom abstract types act *exactly* as regular abstract types, we can split the introduction of those types from the typing. We propose a simple, compositional *source-to-source* transformation that explicitly introduces special abstract type fields *id*, called *identity tag* in Figure 13. We call *tagged* expressions those resulting from the transformation, so as to distinguish them from *raw* (untagged) expressions. Structures and functors are wrapped inside a two-field structure with its identity tag and the actual value. New (abstract) identity tags are introduced when typing structures and functors, or via an ascription. Conversely, identity tags are shared when aliasing a module.

Controlling the applicativity granularity by a source-to-source transformation allows for a simpler set of typing rules. Besides, it leaves open the choice to apply the transformation so as to obtain OCaml coarse-grain granularity (and abstraction safety), or just stay with the default static equivalence.

Example 4.2.1. *An example of the source-to-source transformation. For readability, the result of the transparent ascriptions has been inlined to display the equality of identity types.*

Source code

```
1 module X = struct_A type t = A.t end
2 module X' = (X < sig end)
3 module F = (Y : sig_A end) → (Y < sig end)
```

Transformed code

```
1 module X = struct_B type id = B.id module Val = struct_A type t = A.t end end
2 module X' = struct_B type id = B.id module Val = (X.Val < sig end) end
3 module F = (Y : sig_A type id = A.id module Val : sig end end) →
4   struct_A type id = Y.id module Val = (Y.Val < sig end) end
```

4.2.2 Derived typing system

The source-to-source transformation could also be inlined, introducing phantom identity types during type-checking, as in *F-ing* (Rossberg et al. [2014]). This would affect the rules that "create" new modules, i.e., the rules for structures and functors. In the following, we write the tagged structures with a pair for sake of conciseness:

$$(\alpha_{id}, \mathcal{C}) \triangleq \mathsf{sig\ type}\ id = \alpha_{id} \quad \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end}$$

We get the following derived rules for typing of module expressions:

$$\textsc{M-Typ-Mod-Struct}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\diamondsuit}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\diamondsuit}\alpha_{id}, \overline{\alpha}.(\alpha_{id}, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end})}$$

$$\textsc{M-Typ-Mod-AppFct}\ \frac{\Gamma \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathtt{M} : \exists^{\triangledown}\overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^{\triangledown}\alpha_{id}, \overline{\beta'}.(\alpha_{id}, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right])}$$

$$\textsc{M-Typ-Mod-GenFct}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : \exists^{\triangledown}\alpha_{id}.(\alpha_{id}, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C})}$$

Signature elaboration would be modified similarly, introducing parameterized abstract types for structures and functors. The rest of the type system would not be affected, only dealing with tagged signatures $(\alpha_{id}, \mathcal{C})$ rather than plain signatures $\mathcal{C}$ in some of the rules. As explained before, the identity tags act exactly as normal type fields. Therefore, subtyping on identity tags would be restricted to type equivalence, as it is already the case for other type fields.

4.2.3 Property of identity tags

Identity tags are introduced for new modules, then shared when a module is aliased, either directly or via a transparent ascription. Therefore, if two modules share the same identity tag, they *originate* from a common ancestor with a better signature, as stated by the following theorem:

Theorem 1: Identity tags

Two module expressions that share the same identity tag *originate* from a common ancestor with a more precise signature. Namely,

$$\Gamma \vdash [\![\mathtt{M}_1]\!] : \mathsf{sig\ module}\ Val : \mathcal{C}_1\ \ \mathsf{type}\ id = \tau\ \mathsf{end}$$

$$\Gamma \vdash [\![\mathtt{M}_2]\!] : \mathsf{sig\ module}\ Val : \mathcal{C}_2\ \ \mathsf{type}\ id = \tau\ \mathsf{end}$$

implies

$$\exists \mathcal{C}_0, \quad \Gamma \vdash \mathcal{C}_0 < \mathcal{C}_1 \quad \wedge \quad \Gamma \vdash \mathcal{C}_0 < \mathcal{C}_2$$

That is, $\mathtt{M}_1$ and $\mathtt{M}_2$ originate from a common module, of signature $\mathcal{C}_0$ that subsumes both $\mathcal{C}_1$ and $\mathcal{C}_2$.

The rest of this section is dedicated to the proof of this theorem. First, we consider a slightly modified system called $\mathsf{M}^{\omega}_{<:}$ based on F^{ω} extended with bounded quantification. We identify M^{ω} types and M^{ω} signatures and use τ and $\mathcal{C}$ interchangeably in this section. The system $\mathsf{M}^{\omega}_{<:}$ is built from M^{ω} as follows:

1. We extend the quantifiers $\exists^{\diamond}$, $\forall$, and λ to support bounded quantification for abstract types that serve as identities (the other types being bound by the top bound $\top$).

2. We modify the typing and subtyping rules accordingly. Omitting the rules that either do not feature bounds or simply thread them from the premise to the conclusion, only three typing rules are affected, and now feature an additional subtyping condition in their premises (yellow background is used to emphasize the differences):

MS-Typ-Sig-Con
$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} \mathbin{<} \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

MS-Typ-Mod-Ascr
$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} \mathbin{<} \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown}(\overline{\alpha} \mathbin{<} \overline{\tau}').\mathcal{C}'}$$

MS-Typ-Mod-AppApp
$$\frac{\begin{array}{c}\Gamma \vdash P : \forall(\overline{\alpha} \mathbin{<} \overline{\tau}').\mathcal{C}_a \to \mathcal{C} \\ \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'\end{array}}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

3. We modify the typing rules for introducing abstract types (MS-Typ-Decl-TypeAbs and MS-Typ-Bind-AbsType) by distinguishing identity tags (MS-Typ-Decl-TypeAbsId and MS-Typ-Bind-AbsTypeId) from other abstract types. While abstract types are simply bound by $\top$, identity tags are bound by the signature of the associated value.

MS-Typ-Decl-TypeAbs
$$\frac{t \neq id}{\Gamma \vdash_A (\mathtt{type}\, t = A.t) : \lambda(\alpha < \top).(\mathtt{type}\, t = \alpha)}$$

MS-Typ-Decl-TypeAbsId
$$\frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathtt{type}\, id = A.id) : \lambda(\alpha \mathbin{<} \mathcal{C}).(\mathtt{type}\, id = \alpha)}$$

and, similarly:

MS-Typ-Bind-AbsType
$$\frac{t \neq id}{\Gamma \vdash_A (\mathtt{type}\, t = A.t) : \exists^{\diamond}(\alpha < \top).(\mathtt{type}\, t = \alpha)}$$

MS-Typ-Bind-AbsTypeId
$$\frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathtt{type}\, id = A.id) : \exists^{\diamond}(\alpha \mathbin{<} \mathcal{C}).(\mathtt{type}\, id = \alpha)}$$

4. Subtyping is extended with a new rule MS-Sub-Bound for subtyping applied higher-order bound variables (MS-Sub-Bound-Star is just a particular case of MS-Sub-Bound):

MS-Sub-Bound
$$\frac{\varphi < \lambda(\overline{\alpha} < \overline{\tau}').\mathcal{C} \in \Gamma \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (\varphi\, \overline{\tau}) < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

MS-Sub-Bound-Star
$$\frac{\alpha < \mathcal{C} \in \Gamma}{\Gamma \vdash \alpha < \mathcal{C}}$$

Besides, the two following rules for subtyping between functors are also extended with additional premises to ensure the correct instantiation of abstract types:

$$\textsc{MS-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} < \overline{\tau} \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau_1}] \qquad \Gamma, \overline{\alpha} < \overline{\tau} \vdash \overline{\tau_1} < \overline{\tau'}}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} < \overline{\tau}).\mathcal{C} < () \to \exists^{\blacktriangledown}(\overline{\alpha}' < \overline{\tau'}).\mathcal{C}'}$$

$$\textsc{MS-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \mathcal{C}'_a < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau_1}] \qquad \Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau_1}] < \mathcal{C}' \qquad \Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \overline{\tau_1} < \overline{\tau}}{\Gamma \vdash \forall(\overline{\alpha} < \overline{\tau}).\mathcal{C}_a \to \mathcal{C} < \forall(\overline{\alpha}' < \overline{\tau}').\mathcal{C}'_a \to \mathcal{C}'}$$

Crucially, subtyping in $\mathsf{M}^\omega_{<:}$ remains transitive.

The proof then proceeds in two steps:

1. We first show that $\mathsf{M}^\omega_{<:}$ maintains a stronger notion of wellformedness called *identity wellformedness*: if $\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ \mathit{Val} : \mathcal{C}\ \mathsf{end}$ then $\Gamma \vdash \tau < \mathcal{C}$ (**1**). To do so, we reinforce wellformedness for identity tags by changing the rule to

$$\frac{\Gamma \vdash \mathcal{C} : \mathsf{wf} \qquad \Gamma \vdash \tau < \mathcal{C}}{\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ \mathit{Val} : \mathcal{C}\ \mathsf{end} : \mathsf{wf}}$$

   This defines a stronger wellformedness judgment $\Gamma \vdash \mathcal{C} : \mathsf{wf}_{id}$. We show that $\mathsf{M}^\omega_{<:}$ typing judgments for modules and signatures always produce signatures that preserves identity-tag wellformedness. That is, assuming $\vdash \Gamma : \mathsf{wf}_{id}$, then if either $\Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} < \overline{\mathcal{C}}).\mathcal{C}'$ or $\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit}(\overline{\alpha} < \overline{\mathcal{C}}).\mathcal{C}'$ holds, then $\Gamma, \overline{\alpha} < \overline{\mathcal{C}} \vdash \mathcal{C}' : \mathsf{wf}_{id}$ also holds.

   Therefore, we may restrict $\mathsf{M}^\omega_{<:}$ derivations to use the identity-tag wellformedness invariant as long as we start with an identity-tag wellformed environment. By inversion of wellformedness, this ensures the invariant (1).

2. Using the previous invariant, we then show that typability in M^ω implies typability in $\mathsf{M}^\omega_{<:}$.

The proof of the first step proceeds by a simple induction over the typing derivation: identities are either introduced fresh, in which case the bound is equal to the signature of the corresponding *Val*-field, or obtained via subtyping, in which case we use transitivity of subtyping.

The rest of this section is dedicated to the proof of the second step. The only differences between the original and the enhanced typing systems are the addition of subtyping bounds and subtyping relations between the bounds. The core of the proof is to show that these are actually not restrictive, which follows from two key facts: (1) subtyping is only done with a right-hand side signature that *comes from a source signature*, i.e., that is the result of typing a source signature; and (2) such signatures always contain the bounds of their identity tags in at least one positive occurrence.

Properties of elaborated signatures An M^ω signature that is the elaboration of a source signature (referred to as TSS for *Tagged Source Signature* in the following) has actually a stronger property than being just identity-wellformed. In a TSS, the bound of an identity tag α is *exactly* the signature $\mathcal{C}$ of the first module occurrence with an identity tag α, and therefore, the bound always appears *explicitly* in the signature. By contrast, in signatures that are just identity-wellformed (like the result of inference), there might be no module with the same signature as the bound, but only supertypes.

First-order example Before diving into the proof by induction, we consider the typing of a basic source signature $\mathtt{S}$:

$$\Gamma \vdash \mathtt{S} : \lambda(\alpha < \mathcal{C}_\alpha).\mathcal{C}$$

There must be a subterm of $\mathcal{C}$ at a positive occurrence that is equal to:

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_\alpha\ \mathsf{type}\ \mathit{id} = \alpha\ \mathsf{end}$$

When subtyping this signature with another one in the enhanced system,

$$\Gamma \vdash \lambda(\alpha < \mathcal{C}_\alpha).\mathcal{C} < \lambda(\beta < \mathcal{C}_\beta).\mathcal{C}'$$

there is a new subtyping check between the bounds:

$$\Gamma \vdash \mathcal{C}_\beta < \mathcal{C}_\alpha \tag{1}$$

Whenever the subtyping between $\mathcal{C}'$ and $\mathcal{C}$ succeeded in M^ω, $\mathcal{C}'$ features a subterm of the form

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_0\ \mathsf{type}\ \mathit{id} = \beta\ \mathsf{end}$$

By subtyping, we have $\Gamma \vdash \mathcal{C}_0 < \mathcal{C}_\alpha$. Thanks to the invariant of the first part of the proof, we know that $\Gamma \vdash \mathcal{C}_\beta < \mathcal{C}_0$. Transitivity of subtyping ensures (1). Hence subtyping also succeeds in $\mathsf{M}^\omega_{<:}$.

Proof by induction We prove by induction over the typing derivation that typing and subtyping in M^ω implies typing and subtyping in $\mathsf{M}^\omega_{<:}$. Cases for unchanged rules, or rules that just thread the bounds from the premise to the conclusion are immediate. The only interesting cases are the three typing rules and two subtyping rules shown above that have an additional premise. In each case, we prove that this additional premise is actually implied by the other premises.

Typing M-Typ-Sig-Trans. The M^ω derivation ends with the following rule.

M-Typ-Sig-Trans

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

For simplification of presentation, we assume that $\overline{\alpha}$ is a single variable α. We show that we can rebuild a $\mathsf{M}^\omega_{<:}$ derivation:

MS-Typ-Sig-Con

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda(\alpha < \tau').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\alpha \mapsto \tau]\ (\mathbf{2}) \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \tau < \tau'\ (\mathbf{3})}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\alpha \mapsto \tau]}$$

All the premises but (3), hence including (2), follow by induction hypothesis. The remaining goal is to show that the additional condition (3) actually follows from (2). For this purpose, we define $[\mathcal{C}]^+$, the *positive* declarations of a signature $\mathcal{C}$, as the flattened list of declarations in strict positive positions inside $\mathcal{C}$ (without entering inside generative functors or module-types), taken in the usual binding order, as follows:

$$\begin{aligned}
[\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}]^+ &= \overline{[\mathcal{D}]^+} && \text{(Structural signature)}\\
[\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}]^+ &= \forall\overline{\alpha}.[\mathcal{C}]^+ && \text{(Applicative functor)}\\
[() \to _\,]^+ &= \varnothing && \text{(Generative functor)}\\
[\mathsf{module}\ X : \mathcal{C}]^+ &= (\mathsf{module}\ X : \mathcal{C}), [\mathcal{C}]^+ && \text{(Submodule declaration)}\\
[\mathcal{D}]^+ &= \mathcal{D} && \text{(Other declarations)}
\end{aligned}$$

Declarations inside applicative functors are universally quantified. A key observation is that a signature in TSS form always contains the bound of its identity type among its positive declarations:

$$\Gamma \vdash \mathtt{S} : \lambda(\alpha < \tau).\mathcal{C} \implies (\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \ \wedge\ \mathsf{type}\ \mathit{id} = \alpha \in [\mathcal{C}]^+$$
$$\Gamma \vdash \mathtt{S} : \lambda(\alpha < \lambda\overline{\beta}.\tau).\mathcal{C} \implies \forall\overline{\beta}.(\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \ \wedge\ \forall\overline{\beta}.\mathsf{type}\ \mathit{id} = (\alpha\overline{\beta}) \in [\mathcal{C}]^+$$

Crucially, all the instantiations we consider feature a $\mathsf{TSS}\ \mathcal{C}'$ on their right-hand side. By construction, subtyping between two signatures $\mathcal{C}$ and $\mathcal{C}'$ implies subtyping between declarations at any positive occurrence in $\mathcal{C}$ and its corresponding declaration in $\mathcal{C}'$. Ignoring applicative functors at first, this would give:

$$\Gamma \vdash \mathcal{C} < \mathcal{C}' \implies \forall \mathcal{D}' \in [\mathcal{C}']^+.\ \exists \mathcal{D} \in [\mathcal{C}]^+.\ \Gamma \vdash \mathcal{D} < \mathcal{D}'$$

But there is a catch: the declarations $\mathcal{D}$ and $\mathcal{D}'$ might be deep inside the signature, and therefore the subtyping between them might be true only in an extended environment Γ'. The correct statement is therefore:

$$\Gamma \vdash \mathcal{C} < \mathcal{C}' \implies \forall \mathcal{D}' \in [\mathcal{C}']^+.\ \exists \mathcal{D} \in [\mathcal{C}]^+, \Gamma'.\ \Gamma' \vdash \mathcal{D} < \mathcal{D}' \wedge \Gamma' < \Gamma$$

More specifically, there exists a declaration in the positive part of $\mathcal{C}$ that is a subtype of the explicit bound of α, which appears in $[\mathcal{C}']^+$. That is,

$$\exists \mathcal{C}_0.\ \Gamma' \vdash \mathsf{module}\ \mathit{Val} : \mathcal{C}_0 < \mathsf{module}\ \mathit{Val} : \tau'$$

This implies $\Gamma' \vdash \mathcal{C}_0 < \tau'$. Using the invariant of the first step of the proof, we get $\Gamma' \vdash \tau < \mathcal{C}_0$, which implies (3) by transitivity of subtyping and weakening of the environment. This argument applies to the three adjusted typing rules. It extends to higher-order declarations with universal quantification only adding universally quantified variables in the typing context Γ.

Typing rules MS-Typ-Mod-Ascr and MS-Typ-Mod-AppApp. The former is exactly the same as the previous case. For the latter, the only change if that the bounded quantification $\lambda(\alpha < \tau').\mathcal{C}'$ is being replaced by $\forall(\alpha < \tau').\mathcal{C}'_a \to \mathcal{C}'$.

Subtyping Rule MS-Sub-Sig-AppFct. The same argument as the one used by typing rules applies, just with an extended context. Restricting again to a single type variable for the sake of readability, we need to rebuild a derivation in $\mathsf{M}^\omega_{<:}$ that ends with:

$$\frac{\Gamma, \alpha' < \tau' \vdash \mathcal{C}'_a < \mathcal{C}_a[\alpha \mapsto \tau_1]\ (\mathbf{4}) \qquad \ldots \qquad \Gamma, \alpha' < \tau' \vdash \tau_1 < \tau\ (\mathbf{5})}{\Gamma \vdash \forall(\alpha < \tau).\mathcal{C}_a \to \mathcal{C} < \forall(\alpha' < \tau').\mathcal{C}'_a \to \mathcal{C}'}\ \textsc{MS-Sub-Sig-AppFct}$$

where all the premises but (5) follow by induction hypothesis. As $\mathcal{C}_a$ is a TSS, we have

$$\mathsf{sig\ type}\ \mathit{id} = \alpha\ \mathsf{module}\ \mathit{Val} : \tau\ \mathsf{end} \ \in\ [\mathcal{C}_a]^+$$

Therefore, since α is not free in τ,

$$\mathsf{sig\ type}\ \mathit{id} = \tau_1\ \mathsf{module}\ \mathit{Val} : \tau\ \mathsf{end} \ \in\ [\mathcal{C}_a[\alpha \mapsto \tau_1]]^+$$

Correspondingly, we must have in $\mathcal{C}'_a$, for some signature σ:

$$\mathsf{sig\ type}\ \mathit{id} = \tau_0\ \mathsf{module}\ \mathit{Val} : \sigma\ \mathsf{end} \ \in\ [\mathcal{C}'_a]^+\ (\mathbf{6})$$

Subtyping component by component, we have, since subtyping in non-variant on type fields in general and on identity tag fields in particular:

$$\Gamma, \alpha' < \tau' \vdash \sigma < \tau \quad \wedge \quad \tau_1 = \tau_0$$

The identity-tag wellformedness invariant of (6) implies:

$$\Gamma, \alpha' < \tau' \vdash \tau_0 < \sigma$$

By transitivity of subtyping we get $\Gamma, \alpha' < \tau' \vdash \tau_1 < \tau$, i.e., (5), as expected.

Rule MS-Sub-Sig-GenFct. We rebuild a derivation of the form

$$\frac{\Gamma, \overline{\alpha} < \overline{\tau} \vdash \mathcal{C} < \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau_1}\right] \ (7) \qquad \Gamma, \overline{\alpha} < \overline{\tau} \vdash \overline{\tau_1} < \overline{\tau}' \ (8)}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} < \overline{\tau}).\mathcal{C} < () \to \exists^{\blacktriangledown}(\overline{\alpha}' < \overline{\tau}').\mathcal{C}'} \ \textsc{MS-Sub-Sig-GenFct}$$

The proof is similar to the previous case where (7) follows by induction and (8) follows from (7) and the fact that $\mathcal{C}'$ is in TSS-form.

This completes the proof.

4.3 Rebuilding Source Signatures

In this section, we present a reverse translation from M^{ω} signatures back into the source syntax, called *anchoring*. It is a partial inverse of signature elaboration. This translation is necessarily incomplete as some inferred signatures cannot be expressed in the less expressive source syntax. Crucially, anchoring relies on identity tags, and therefore assumes that we have tagged source programs as described in Section 4.2 prior to type checking. That is, anchoring translates *tagged* signatures back into source (hence untagged) signatures.

In Section 4.3.1, we detail the three expressiveness gaps of the source syntax, which gives us three *anchoring conditions*, associated with three new typechecking errors when those conditions are violated. We argue that this classification leads to more understandable signature avoidance error messages. In Section 4.3.2, we present an anchoring algorithm. In Section 4.3.3, we state and prove the properties of this algorithm, which shines a new light on the differences in the way type-sharing is expressed between M^{ω} and the source syntax.

4.3.1 The Expressiveness Gaps of the Source Syntax

In this section we introduce three *anchoring conditions*. For each of them, we show examples of an M^{ω}-signature meeting the condition and explain the error when the condition is not met. Those error cases constitute a new form of typechecking error that is finer grained than a simple "avoidance" error. Overall, the anchoring conditions ensure that an M^{ω}-signature is the result of the elaboration of some source signature, with an additional constraint for functor applications. We state three anchoring conditions for pedagogical purposes, but the third one actually subsumes the first two.

For the sake of readability, we do not display all module identities in the examples but only the ones that are relevant.

Abstract Type Fields

Structural information and introduction of types A first key insight is the difference in the source syntax between the declaration of a *manifest* type ($\texttt{type}\, t = \texttt{u}$) and that of an *abstract* type ($\texttt{type}\, t = A.t$). An abstract type declaration $\texttt{type}\, t = A.t$ in a covariant position effectively *creates* a new abstract type (introducing an existential quantifier in M^{ω}) and *adds* a type field t to the signature. By contrast a manifest type definition $\texttt{type}\, t = \texttt{u}$ only states structural information – adding a field t to refer to the existing type $\texttt{u}$.

Therefore, in the source syntax, the structural information (name and position of fields) also determines the scope of abstract types. Conversely, the M^ω syntax *separates* the introduction of new abstract types from the introduction of fields by using explicit quantifiers. In particular, they may mention an abstract type without (or before) having a type declaration that refers to it. Overall, anchoring is possible if the structural and scoping information happen to coincide, i.e., if there is a type declaration $\mathsf{type}\ t = \alpha$ for each type variable α that "introduces" α in the right scope. As detailed in Section 3.1.1, new scopes are introduced by generative functors and module-type definitions. This gives the following first anchoring guideline:

Anchoring Condition 1

The first occurrence of an abstract type variable α must be a type declaration that is: (1) of the form $\mathsf{type}\ t = \alpha$, (2) in a strictly positive position, and (3) in the same scope as the binder of α

If such type declaration exists, it is called the *anchoring point* of α. If one of the three conditions is not met, the M^ω signature cannot be anchored. This leads to the following error cases:

(1) If the first occurrence is not a type field (as on the left-hand side), or if does not contain exactly α (as on the right-hand side):

```
1 ∃▽α. sig val x : α  ... end
```

```
1 ∃▽α. sig type t = α × bool  ... end
```

(2) If the first occurrence is not in a (strictly) positive position, used in a functor parameter for instance:

```
1 ∃◇α. sig module F : (sig type t = α end) → (sig ... end) end
```

(3) The first occurrence is not in the same scope as the binder, either inside a generative functor or a module-type (we detail the treatment of applicative functor later in this section), as in:

```
1 ∃◇α. module G : () → sig type t = α end
2 ∃◇α. module type T = sig type t = α end
```

Related notions This notion is linked to *type locators* of Dreyer [2007b]; Rossberg and Dreyer [2013]: the locator is the original anchoring point of α. A key mechanism of anchoring is to re-discover a new locator, as the original one may have been lost. The notion of anchoring point is also closely related to the notion of *type variable roots* in Rossberg et al. [2014]: the anchoring point of α is a root for α, but the converse is not necessarily true. Type roots are used to show the decidability of type-checking, but do not have to be the first occurrence.

Example 4.3.1. *In the following M^ω signature (left-hand side), the first occurrence of α is a type declaration that can serve as an anchoring point. Therefore, we can anchor the signature to the right-hand-side source signature:*

```
1 ∃◇α. module M : sig
2   type t = α
3   val x : α
4   type u = α × int
5 end
```

```
1 module M : sig_A
2   type t = A.t
3   val x : A.t
4   type u = A.t × int
5 end
```

The type declaration $\mathtt{type}\, t = A.t$ (line 2, right-hand side) is the *anchoring point* of α.

Module Identities

The source syntax can only express identity sharing between modules via *transparent* signatures ($= P < \mathtt{S}$). However, a transparent signature ($= P < \mathtt{S}$) is wellformed only if the signature of the path P is a subtype of $\mathtt{S}$. This forces all modules sharing the (same) identity of P to have a signature that is a subtype of (the signature of) the module at P. Therefore, identity sharing is tied with *subtyping conditions* between the signature of each occurrence and the signature of the first occurrence. By contrast, M^ω signatures can express identity sharing regardless of the associated signatures.

Anchoring Condition 2

The first occurrence of an identity tag variable α_{id} must be at a module binding of the form $\mathsf{module}\ X : (\alpha_{id}, \mathcal{C})$, in a strictly positive position, and in the same scope as the binder. Besides, all further occurrences of α_{id}, of the form $(\alpha_{id}, \mathcal{C}')$, must be such that there is subtyping between $\mathcal{C}$ and $\mathcal{C}'$.

As a consequence of this condition, anchoring can fail even if all variables have an anchoring point as their first occurrence: we might discover later in the signature that subtyping conditions are not met. In addition to the three error cases that are similar to type variables, we have a new error case: if one of the following occurrences of the identity tag variable is associated with a signature that is not a subtype of the one at the anchoring point:

```
1 ∃^◇ α_id. module M : sig
2     module X_1 : (α_id, sig  end)
3     module X_2 : (α_id, sig type t = int end)
4 end
```

Here, the signature of X_1 is not a subtype of the one of X_2: $\mathsf{sig}\ \ \mathsf{end} \not< \mathsf{sig\ type}\ t = \mathtt{int\ end}$

Example 4.3.2. *In the following* M^ω*-signature (left-hand-side), the first module with the identity tag serves as the anchoring point for the identity variable* α_{id}*.*

```
1 ∃^◇ α_id, α. module M : sig
2     module X_1 : (α_id, sig type t = α end)
3     module X_2 : (α_id, sig  end)
4 end
```

```
1 module M : sig_A
2   module X_1 : sig_A type t = A.t end
3   module X_2 : (= X_1 < sig  end)
4 end
```

All other occurrences of α_{id} *(here, only at the module declaration of* X_2*) validate the subtyping condition, as we have:* $\mathsf{sig}\ \ \mathsf{end} < \mathsf{sig\ type}\ t = \alpha\ \mathsf{end}$

Higher-order Abstract Types

Let us consider an abstract type t inside an applicative functor:

```
1 module F : (Y : S) → sig_A ... type t = A.t ... end
```

This type field t is reachable by a path with a functor application, of the form $F(X).t$. Therefore, the type does not act exactly as a higher-order type, but is restricted to a certain *domain* that is limited by the signature of the parameter S. Yet M^ω uses a normal higher-order abstract variable φ to model this type declaration, and φ can be applied to any type argument (without restriction). For anchoring, we need to replace occurrences of φ by applicative paths of the form $F(X).t$, which will introduce subtyping conditions (between the signature of X and S).

Anchoring Condition 3 (1/2)

The first occurrence of a higher-order abstract type φ must be a type declaration that is: (1) of the form $\mathsf{type}\ t = \varphi(\overline{\alpha})$, (2) where $\overline{\alpha}$ is exactly the set of universally quantified variables in the environment in the current scope, (3) in a strictly positive position and (4) in the same scope as the binder. Besides, all further occurrences of φ, of the form $\varphi(\overline{\tau})$, must be the result of the elaboration of a (well-formed) path, of the form $F(X).t$.

Again, this introduces new error cases when the conditions are not met. If a higher-order abstract type does not have an anchoring point, we say that we have a "lost functor" anchoring error. Besides the three conditions (first occurrence, strict positivity, same scope) that are similar to base type variables, we have:

1. If the first occurrence of φ is a type declaration of the form $\mathsf{type}\ t = \varphi(\overline{\tau})$, where $\overline{\tau}$ is not exactly the set of universally quantified variables, it means that φ *originates* from a functor with a different arity, and we have an error:

```
1 ∃^◇ φ, α_id. sig
2   module X : sig type t = bool end
3   module F : ∀α.(α, sig  end) → sig type t = φ(α, α_id) end
4   type t = φ(α_id, α_id)
5 end
```

   Here, the variable φ was introduced by a functor that took two parameters, and it only remains a partial application $\varphi(\alpha, \alpha_{id})$ in the signature. Even if the rest of the signature only uses φ in a way that could be emulated using F, i.e., the last field could technically be anchored as $\mathsf{type}\, t = F(X).t$, we refuse to do so and throw an error, as it is a form of over-abstraction.

2. If we have a suitable anchoring point, there might be occurrences further down the signature that cannot be expressed as paths to the anchoring point, due to the subtyping conditions.

```
1 ∃^◇ φ, α_id. sig
2   module F : ∀α.(α, sig val x : int end) → sig type t = φ(α) end
3   module X : (α_id, sig val x : bool end)
4   type t = φ(α_id)
5 end
```

   Here, the type declaration cannot be anchored, as the path $F(X).t$ would be ill-formed. Indeed, the signature of X is not a subtype of the functor's parameter, as they differ on the type of the value field x. Such situation can appear by inference, when φ was introduced by a functor that was *more general* than F, i.e., which parameter's signature did not contain a field x.

Relation between the three anchoring conditions The condition (1) is a sub-case of the condition (3), restricted to base types (of the base kind $\star$). The subtyping constraint that is introduced by condition (2) is similar to the one condition (3). This comes from the fact transparent ascription can be encoded as functor application: for each signature $\mathtt{S}$, we could introduce a dummy functor $F_{\mathtt{S}} : (Y : \mathtt{S}) \to Y$. Each transparent ascription ($= P < \mathtt{S}$) could be obtained as the result of the application $F_{\mathtt{S}}(P)$.

Example 4.3.3. *In the following* M^ω*-signature (left-hand side), the anchoring conditions are met. All occurrences of* φ *can be anchored as paths to the anchoring point. Those paths can contain functor parameters, as displayed by* F_2*.*

```
1  ∃▽ φ, α_id. module M : sig
2    module F_1 :
3      ∀α_id.(α_id, sig  end) →
4        sig type t = φ(α_id) end
5    module X : (α_id, sig val x : int end)
6    type t = φ(α_id)
7    module F_2 :
8      ∀α_id.(α_id, sig val x : bool end) →
9        sig type t = φ(α_id) end
10 end
```

```
1  module M : sig_A
2    module F_1 :
3      (Y : sig  end) →
4        sig_B type t = B.t end
5    module X : sig val x : int end
6    type t = A.F_1(A.X).t
7    module F_2 :
8      (Y : sig val x : bool end) →
9        sig type t = A.F_1(Y).t end
10 end
```

Disabling functor applications "out of thin air" As discussed in Section 2.3.3, we want to prevent the anchoring from *inventing* paths with functor applications that never appeared in the source, just for referring to abstract types that have lost their original path. Doing so would be quite surprising, if not misleading, as it would suggest a computation that will never happen. Therefore, we extend the third anchoring condition:

Anchoring Condition 3 (2/2)

The anchoring point of a higher-order type should be *original*, in the functor that introduced it or in an alias of the functor that introduced it.

To distinguish *original anchoring points*, we will need to slightly instrument the typing rules, as this information cannot be reconstructed just from types.

4.3.2 The Anchoring Process

In this section we present an algorithm that produces a source signature given an M^ω one, when the anchoring conditions are met. For pedagogical purposes, it is split in two steps: we first translate M^ω signatures into tagged source signatures, before removing tags to obtain source signatures. Both steps are presented as relations, although they are deterministic. We first explain some instrumentation added to the typing judgment: scope barriers, arity delimiters, type variables binder indicator, and application marks.

Instrumenting the typing judgment (1/4) First, we extend the grammar of environments with scope *barriers*: we write $\Gamma \cdot \Gamma'$ for an environment that behaves as Γ, Γ' but with a barrier between Γ and Γ' and let Δ range over environments without barriers. Hence, by writing $\Gamma \cdot \Delta$, we mean that Δ is the part of the environment right after the rightmost barrier. This is used to indicate scopes (adding a barrier) and prevent anchoring of types that have been introduced in a larger scope. Barriers are introduced in the context by typing rules that open scopes:

$$\textsc{M-Typ-Sig-GenFct}\ \frac{\Gamma \cdot \vdash \mathtt{M} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Decl-ModType}\ \frac{\Gamma \cdot \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

$$\textsc{M-Typ-Mod-GenFct}\ \frac{\Gamma \cdot \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Bind-ModType}\ \frac{\Gamma \cdot \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

Instrumenting the typing judgment (2/4) We also instrument Rule M-Typ-Mod-AppFct to mark skolemization steps, writing $\beta'\langle\overline{\alpha}\rangle$ instead of $\beta'(\overline{\alpha})$ but to mean the same, so that anchoring may pattern-match on a list of lists of arguments rather than on a flat list:

$$\text{M-Typ-Mod-AppFct}\quad \frac{\Gamma \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathtt{M} : \exists^\nabla \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^\nabla \overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'\langle\overline{\alpha}\rangle}\right]}$$

This makes the treatment of arity of functors easier.

Instrumenting the typing judgment (3/4) When adding new variables in the environment, we add a flag when they come from an universal quantification. We write $!\overline{\alpha}$ to indicate that $\overline{\alpha}$ were universally quantified and we write $?\overline{\alpha}$ otherwise (for existential or lambda quantification). This enables the definition of an operator $\mathsf{args}(\Delta)$, that returns the list (of lists) of universally quantified variables. For the sake of readability, the flags are written only when relevant for the context. Alternatively, we could also have identified universally quantified variables by the fact that they immediately precede functor parameters in Δ. We used flags because they also simplify the presentation of the proofs in Section 4.3.3.

Instrumenting the typing judgment (4/4) Finally, we modify the typing rule for functor application M-Typ-Mod-AppApp to *mark* higher-order abstract types, in order to identify *original* anchoring points: type declarations obtained by a functor application are marked as not original, while aliasing a functor copies its type declaration unmarked. Technically, this is achieved by using a syntactic mark $\tau^\dagger$ on types, which can be seen as the introduction of a postfixed constant $\dagger$ that behaves as $\lambda\alpha.\alpha$. That is, $\tau^\dagger$ syntactically differ from τ but really means τ. Marks are ignored by subtyping rules, which may freely erase them. We write $\mathcal{C}^\dagger$ for the signature $\mathcal{C}$ where all type declarations $\mathsf{type}\ t = \tau$ of the structure and substructures have been rewritten into $\mathsf{type}\ t = \tau^\dagger$—but the marking does not go inside the body of functors nor inside module-types. Therefore, we only change the resulting signature of the rule M-Typ-Mod-AppApp to a marked signature $\mathcal{C}^\dagger[\overline{\alpha} \mapsto \overline{\tau}]$:

$$\text{M-Typ-Mod-AppApp}\quad \frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash P(P') : \mathcal{C}^\dagger[\overline{\alpha} \mapsto \overline{\tau}]}$$

From M^ω Signatures to Tagged Source Signatures

The algorithm proceeds by visiting the M^ω signature in left-to-right, depth-first order. Along the way, it removes all universal and existential quantifiers from the M^ω signature and replaces occurrences of the corresponding abstract type variables by either a self-reference (at its anchoring point) or a path referring to its anchoring point. An *anchoring map* θ from M^ω types τ to qualified types $P.t$ is built and updated during the visit. The algorithm is defined by mutually recursive judgments:

- **Environment anchoring** $\Gamma \hookrightarrow \theta$ checks that θ is a correct anchoring map for Γ.
- **Signature anchoring** $\Gamma\,;\theta \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathtt{S} : \theta$, given a path P (which is actually shallow, i.e., taking one of the three forms Y, $A.X$, or $A.Val(Y)$), translates the M^ω-signature $\mathcal{C}$ into a tagged source signature $\mathtt{S}$ and produces a (possibly empty) local anchoring map θ of the abstract types anchored in $\mathtt{S}$, prefixed by P. We also define declaration anchoring $\Gamma\,;\theta \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta$, with a self-reference A in place of the path P.
- **Local anchoring** $\Gamma;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_\,)$ just checks that signature $\mathcal{C}$ can be translated into $\mathtt{S}$ producing a local map of domain $\overline{\alpha}$ that is ignored afterwards.

- **Type anchoring** $\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}$ translates the M^ω type τ to a source type $\mathtt{u}$, replacing each M^ω variable by a path to its anchor, as described in θ.

The whole set of rules is given in Figure 14. The key rules are those that extend, update, or use the anchoring map to reconstruct source type expressions. We start with a simple example:

Example 4.3.4. *To illustrate the anchoring process, we consider the M^ω signature on the left-hand side. At each line, we give the corresponding anchoring map, used to produce the source signature on the right-hand side.*

```
1 ∃▽α, β. sig
2   type t = α
3   module X : sig
4     type u = β
5     val x : β × α
6   end
7   val y : β × α
8 end
```

$\varnothing$
$(\alpha \mapsto A.t)$
$(\alpha \mapsto A.t)$
$(\alpha \mapsto A.t) \uplus (\beta \mapsto B.u)$
$(\alpha \mapsto A.t) \uplus (\beta \mapsto B.u)$
$(\alpha \mapsto A.t) \uplus (\beta \mapsto A.X.u)$
$(\alpha \mapsto A.t) \uplus (\beta \mapsto A.X.u)$

```
1 sig_A
2   type t = A.t
3   module X : sig_B
4     type u = B.t
5     val x : B.t × A.t
6   end
7   val y : A.X.u × A.t
8 end
```

The anchoring map is extended at line (2) and (4) when the anchoring points for α and β are found. The map is updated at line (6) when exiting the submodule X: the path for the anchoring point of β is prefixed by X.

Anchoring points A new anchoring point is introduced when reaching a type declaration of the form $\mathsf{type}\ t = \alpha$ where α is not anchored yet, i.e., not in the domain of θ. A simplified rule for first-order types is:

$$\frac{\alpha \notin \mathsf{dom}(\theta) \qquad \alpha \in \Delta \qquad \mathsf{args}(\Delta) = \varnothing}{\Gamma \cdot \Delta\,;\theta \vdash \mathsf{type}\ t = \alpha \overset{A}{\hookrightarrow} \mathtt{type}\ t = A.t : (\alpha \mapsto A.t)}$$

We ensure that the type α has been introduced after the left-most barrier, by requiring $\alpha \in \Delta$, and check that the type declaration has not been made inside an applicative functor by requiring that the environment Δ contains no universally quantified types. The check for positivity is made when exiting a functor definition, it is not visible here. We then return the singleton map $(\alpha \mapsto A.t)$. The general version of the rule A-Decl-Anchor considers a declaration for a possibly higher-order *unmarked* type expression φ:

A-Decl-Anchor
$$\frac{\varphi \notin \mathsf{dom}(\theta) \qquad \varphi \in \Delta \qquad \mathsf{args}(\Delta) = \overline{\alpha_1}\,;\ \ldots\ \overline{\alpha_n}}{\Gamma \cdot \Delta\,;\theta \vdash \mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \overset{A}{\hookrightarrow} \mathtt{type}\ t = A.t : (\varphi \mapsto A.t)}$$

This rule only applies when φ is both unmarked and applied to exactly the sequence $\mathsf{args}(\Delta)$ of abstract types in Δ (which necessarily follow φ). Anchoring fails if one of the conditions does not hold. The process could be made more permissive or more restrictive by tweaking this rule.

Updates The anchoring map θ is *updated* in the two places where access paths to types must be changed as we exit scopes: (1) in Rule A-Sig-StrPath, locally anchored abstract types are made available through the path P and (2) in Rule A-Sig-FctApp, paths to anchored types of θ are point-wise abstracted over the functor parameter Y in the returned map $\lambda Y.\theta$. The list $\overline{\alpha}$ is marked as existentially quantified in the left-hand side premise (when anchoring $\mathcal{C}_a$)

A-Env-Decl
$$\frac{\Gamma\,;\theta \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta'}{\Gamma, A.\mathcal{D} \hookrightarrow \theta \uplus \theta}$$

A-Env-Arg
$$\frac{\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta_a) = \overline{\alpha}}{\Gamma, \overline{\alpha}, (Y : \mathcal{C}) \hookrightarrow \theta \uplus \theta_a}$$

A-Env-Abs
$$\frac{\Gamma \hookrightarrow \theta}{\Gamma, \overline{\alpha} \hookrightarrow \theta}$$

(a) Anchoring of environment

A-Sig-StrPath
$$\frac{\Gamma\,;\theta \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta \qquad A \notin \Gamma}{\Gamma\,;\theta \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \overset{P}{\hookrightarrow} \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : \theta[A \mapsto P]}$$

A-Sig-StrNone
$$\frac{\Gamma\,;\theta \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta \qquad A \notin \Gamma \qquad \mathsf{dom}(\theta) = \overline{\alpha}}{\Gamma\,;\theta \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \hookrightarrow \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : (\overline{\alpha} \mapsto \_)}$$

A-Sig-FctGen
$$\frac{\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_)}{\Gamma\,;\theta \vdash () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} \overset{A.Val}{\hookrightarrow} () \to \mathtt{S} : \varnothing}$$

A-Sig-FctApp
$$\frac{\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta_a) = \overline{\alpha} \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \overset{A.Val(Y)}{\hookrightarrow} \mathtt{S} : \theta}{\Gamma\,;\theta \vdash \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \overset{A.Val}{\hookrightarrow} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$

(b) Anchoring of signatures

A-Decl-Val
$$\frac{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}{\Gamma\,;\theta \vdash \mathsf{val}\ x : \tau \overset{A}{\hookrightarrow} (\mathtt{val}\ x : \mathtt{u}) : \varnothing}$$

A-Decl-Anchor
$$\frac{\varphi \notin \mathsf{dom}(\theta) \qquad \varphi \in \Delta \qquad \mathsf{args}(\Delta) = \overline{\alpha_1}\,;\ \ldots\ \overline{\alpha_n}}{\Gamma \cdot \Delta\,;\theta \vdash \mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \overset{A}{\hookrightarrow} \mathtt{type}\ t = A.t : (\varphi \mapsto A.t)}$$

A-Decl-Type
$$\frac{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}{\Gamma\,;\theta \vdash \mathsf{type}\ t = \tau \overset{A}{\hookrightarrow} \mathtt{type}\ t = \mathtt{u} : \varnothing}$$

A-Decl-Mod
$$\frac{\Gamma\,;\theta \vdash \mathcal{C} \overset{A.X}{\hookrightarrow} \mathtt{S} : \theta}{\Gamma\,;\theta \vdash \mathsf{module}\ X : \mathcal{C} \overset{A}{\hookrightarrow} \mathtt{module}\ X : \mathtt{S} : A.\theta}$$

A-Decl-Empty
$$\Gamma\,;\theta \vdash \varnothing \overset{A}{\hookrightarrow} \varnothing : \varnothing$$

A-Decl-ModType
$$\frac{\Gamma, \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_)}{\Gamma\,;\theta \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \overset{A}{\hookrightarrow} \mathtt{module\ type}\ \ T = \mathtt{S} : \varnothing}$$

A-Decl-Seq
$$\frac{\Gamma\,;\theta \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta_1 \qquad \Gamma, A.\mathcal{D}\,;\theta \uplus \theta_1 \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta_2}{\Gamma\,;\theta \vdash \mathcal{D}, \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \mathtt{D}, \overline{\mathtt{D}} : \theta_1 \uplus \theta_2}$$

(c) Anchoring of declarations

A-Type-Application
$$\frac{\begin{array}{c}\tau = \varphi\langle\tau_1 \ldots\rangle \ldots \langle\tau_n \ldots\rangle \qquad \theta(\varphi) = \lambda Y_k. \ldots . \lambda Y_n.\ P.t \\ \forall i \in [\![k, n]\!]\,.\ \Gamma\,;\theta \vdash \tau_i \hookrightarrow P_i.id \qquad \mathtt{u} = \theta(\varphi)(P_k) \ldots (P_n) \qquad \Gamma \vdash \mathtt{u} : \tau\end{array}}{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}$$

(d) Anchoring of types

Figure 14: Anchoring rules

and universally quantified in the right-hand side one (when anchoring $\mathcal{C}$).

$$\textsc{A-Sig-StrPath}\quad \frac{\Gamma\,;\theta \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta \qquad A \notin \Gamma}{\Gamma\,;\theta \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \overset{P}{\hookrightarrow} \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : \theta[A \mapsto P]}$$

$$\textsc{A-Sig-FctApp}\quad \frac{\Gamma \cdot ?\overline{\alpha}\,;\theta \vdash \mathcal{C}_a \hookrightarrow \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta) = \overline{\alpha} \qquad \Gamma, !\overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \xhookrightarrow{A.Val(Y)} \mathtt{S} : \theta}{\Gamma\,;\theta \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{A.Val} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$

By contrast, the anchoring map of the body of a generative functor is thrown away (Rule A-Sig-FctGen), as generative functors cannot appear in paths, [1] and a barrier is added in the premise, as the body cannot capture types defined outside of the functor.

Path resolution Finally, the anchoring map is used for anchoring an M^{ω}-types τ into a qualified type – which requires finding a suitable path to access the anchoring point – by the following rule:

$$\textsc{A-Type-Application}\quad \frac{\begin{array}{c}\tau = \varphi\langle\tau_1 \dots\rangle \dots \langle\tau_n \dots\rangle \qquad \theta(\varphi) = \lambda Y_k. \dots \lambda Y_n.\ P.t \\ \forall i \in [\![k, n]\!]\,.\,\Gamma\,;\theta \vdash \tau_i \hookrightarrow P_i.id \qquad \mathtt{u} = \theta(\varphi)(P_k) \dots (P_n) \qquad \Gamma \vdash \mathtt{u} : \tau\end{array}}{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}$$

Here, we consider types stripped of their marks, e.g., by reducing $\varphi^{\dagger}$ to φ. The rule A-Type-Application applies when τ is of the form $\varphi\langle\tau_1 \dots\rangle \dots \langle\tau_n \dots\rangle$ and $\theta(\varphi)$ is a (mathematical) function of the form $\lambda Y_k. \dots \lambda Y_n.P.t$. Types τ_k to τ_n may only be resolved to identity tags $P_k.id$ to $P_n.id$. This rule is designed to allow anchoring of a type τ that is abstract over a certain number of parameters (here, $n - k + 1$) even if τ is actually applied to more parameters (here, n). This comes from the fact that source signatures do not display the depth of enclosing applicative functors. The resulting type $\mathtt{u}$ is the path (resulting from the mathematical application) $\theta(\varphi)(P_k) \dots (P_n)$. However, $\mathtt{u}$ must be re-typechecked to ensure that paths occurring in τ only contain valid functor applications[2] and that it returns the same type as the input type τ.

Example 4.3.5. *We illustrate the anchoring process of a type inside an applicative functor. We give the corresponding anchoring map at the end of the line. For the sake of readability, we do not write the identity of the functor (nor its body) and we write tagged signatures with*

[1] Similarly, Rule A-Decl-ModType only checks for anchorability of the body of a module-type, and then discards the anchoring map, since module-types cannot appear in paths.

[2] Indeed, it can happen that a module `X` is lost, while a transparent ascription `X'` is kept. The types resulting from a functor application `F(X).t` may not be anchorable as `F(X').t` if `X'` lacks certain fields.

pairs.

```
1  ∃▽φ, α_id1, α_id2.sig
2   module F :
3    ∀α_id, α.(α_id, sig type t = α end) →
4     sig type t = φ(α) end
5
6   module X1 : (α_id1, sig type t = int end)
7   module X2 : (α_id2, sig type t = bool end)
8   type t = φ ⟨α_id1, int⟩
9   type u = φ ⟨α_id2, bool⟩
10 end
```

```
∅
∅
(α_id ↦ Y.id, α ↦ Y.Val.t)
(α_id ↦ Y.id, α ↦ Y.Val.t, φ ↦ C.t)
(φ ↦ λY.A.F(Y).t)
(φ ↦ λY.A.F(Y).t, α_id1 ↦ A.X1.id)
(φ ↦ λY.A.F(Y).t, α_id1 ↦ A.X1.id, α_id2 ↦ A.X2.id)
(φ ↦ λY.A.F(Y).t, α_id1 ↦ A.X1.id, α_id2 ↦ A.X2.id)
(φ ↦ λY.A.F(Y).t, α_id1 ↦ A.X1.id, α_id2 ↦ A.X2.id)
```

```
1  sig_A
2    module F : (Y : sig_B type t = B.t end) → sig_C type t = C.t end
3    module X1 : sig_A1 type id = A1.id  module Val : (sig type t = int end) end
4    module X2 : sig_A2 type id = A2.id  module Val : (sig type t = bool end) end
5    type t = A.F(A.X1).t
6    type u = A.F(A.X2).t
7  end
```

We have the following steps:

3. *When anchoring the body of the functor, we have an anchoring of the universally quantified variables α_{id}, α.*
4. *Then, an anchoring point for φ is found, initially bound to the local name $C.t$.*
5. *When exiting the functor, the anchoring of α_{id}, α is thrown, as the variables are no longer accessible, and the anchoring of φ is updated to a parameterized path $\lambda Y.A.F(Y).t$*
6. *The identity tag α_{id_1} is anchored to $X_1.id$*
7. *The identity tag α_{id_2} is anchored to $X_2.id$*

Finally, the anchoring map is used at lines (8) and (9) to rebuild the qualified types.

Untagging

The first step of anchoring returns a tagged source signature $\mathtt{S}$. It remains to remove the tags, i.e., to return a signature $\mathtt{S}'$ with the `id` and `Val` fields stripped of $\mathtt{S}$, recursively, but expressing the same sharing using transparent signatures. This is defined as a judgment $\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}'$. The two interesting rules are for untagging structural signatures:

$$\frac{\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}'}{\Gamma \vdash \mathsf{Tag}\,\{\mathtt{S}\} \hookrightarrow \mathtt{S}'}\ \textsc{U-Sig-Fresh} \qquad \frac{\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}' \quad \Gamma \vdash \mathtt{S}' : \mathcal{C}' \quad \Gamma \vdash P : \mathcal{C} \quad \Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash \mathtt{sig}_A\ \mathtt{module}\ \mathit{Val} : \mathtt{S}\ \mathtt{type}\ \mathit{id} = P.\mathit{id}\ \mathtt{end} \hookrightarrow (= P < \mathtt{S}')}\ \textsc{U-Sig-Con}$$

When the identity type declaration is an *abstract* type declaration $\mathsf{Tag}\,\{\mathtt{S}\}$ (Rule U-Sig-Fresh), i.e., of the form $\mathtt{sig}_A\ \mathtt{type}\ \mathit{id} = P.\mathit{id}\ \mathtt{module}\ \mathit{Val} : \mathtt{S}\ \mathtt{end}$, the identity of the module is *fresh*, hence the anchored signature of the value is returned directly. Otherwise (Rule U-Sig-Con), the identity type declaration is concrete, i.e., of the form $P.\mathit{id}$; that is, the signature of a module that shares its identity with the module P. We retrieve the M^ω-signature $\mathcal{C}$ of the module P and check that it is a subtype of the M^ω-signature $\mathcal{C}'$ of the untagging $\mathtt{S}'$ of $\mathtt{S}$, so as to ensure that the transparent signature $(= P < \mathtt{S}')$ to be returned is valid.

The other rules are given in only remove the access to *Val*-fields and inductively call untagging.

$$\textsc{U-Sig-Fresh}\ \frac{\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}'}{\Gamma \vdash \mathsf{Tag}\,\{\mathtt{S}\} \hookrightarrow \mathtt{S}'} \qquad \textsc{U-Sig-Con}\ \frac{\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}' \qquad \Gamma \vdash \mathtt{S}' : \mathcal{C}' \qquad \Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash \mathtt{sig}_A\ \mathtt{module}\ \mathit{Val} : \mathtt{S}\ \mathtt{type}\ \mathit{id} = P.\mathit{id}\ \mathtt{end} \hookrightarrow (= P < \mathtt{S}')}$$

$$\textsc{U-Sig-FctA}\ \frac{\Gamma \vdash \mathtt{S}_a \hookrightarrow \mathtt{S}'_a \qquad \Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}'}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S} \hookrightarrow (Y : \mathtt{S}'_a) \to \mathtt{S}'} \qquad \textsc{U-Sig-FctG}\ \frac{\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}'}{\Gamma \vdash () \to \mathtt{S} \hookrightarrow () \to \mathtt{S}'}$$

Figure 15: Untagging rules

4.3.3 Properties of Anchoring

In this section we state and prove the meta-theoretic properties of anchoring.

Anchoring of elaborated signatures Conceptually, anchoring and elaboration of signatures are inverse of each other, which we could write as "Elaboration ∘ Anchoring = Id". There is however a caveat, as typechecking is not injective: several source signatures can express the same type sharing information. Therefore, we may quotient source signatures by the equivalence induced by M^ω typing (using the type equivalence of M^ω). We define the equivalence of source signatures as:

$$\Gamma \vdash \mathtt{S} \approx \mathtt{S}' \ \triangleq\ \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \vdash \mathtt{S}' : \lambda\overline{\alpha}.\mathcal{C}' \ \wedge\ \lambda\overline{\alpha}.\mathcal{C} \equiv \lambda\overline{\alpha}'.\mathcal{C}'.$$

Anchoring produces a signature where all type equalities have been inlined.

Theorem 2: Elaboration ∘ Anchoring = Id

The elaboration of source signatures produces anchorable signatures. Given a source signature $\mathtt{S}$, typing environment Γ and anchoring map θ such that:

$$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \hookrightarrow \theta$$

Then, there exists a source signature $\mathtt{S}'$ and an anchoring map θ' such that:

$$\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S}' : (\overline{\alpha} \mapsto \_\,)$$

The signature $\mathtt{S}'$ might not be syntactically equal to $\mathtt{S}$, but is equivalent to $\mathtt{S}$ (with regard to M^ω typing):

$$\Gamma \vdash \mathtt{S} \approx \mathtt{S}'$$

Elaboration of anchored signatures The other direction would be to show that, starting with a M^ω signature $\mathcal{C}$, finding an anchored signature $\mathtt{S}$ and elaborating it back to M^ω would give a signature $\mathcal{C}'$ that is equivalent to $\mathcal{C}$, the signature we started with. This could be seen as "Anchoring ∘ Elaboration = Id". Yet, there is a subtlety. M^ω signatures can express the fact that they are *inside an applicative functor*, as their abstract types are applied to a certain set of universally quantified type variables. This does not appear in source signatures, which are the same whatever the number of enclosing applicative functors. Therefore, we only have an equivalence "up to *skolemization*".

Theorem 3: Anchoring ∘ Elaboration ≃ Id

Given a M^ω signature $\mathcal{C}$, a source signature $\mathtt{S}$, a typing environment and anchoring maps θ and θ' such that:

$$\Gamma \cdot \Delta \,;\, \theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : \theta' \quad \text{and} \quad \Gamma \hookrightarrow \theta \quad \text{and} \quad \mathsf{dom}(\theta') = \overline{\alpha}$$

Typing back the anchoring gives the original signature, up to re-skolemization (depending on the universally quantified types in the environment).

$$\Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}' \ \wedge\ \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\alpha}(\mathsf{args}(\Delta))\right] = \mathcal{C}$$

Untagging Finally, the untagging phase is not surprising, and is the inverse of tagging. The result relies on a notion of *tag-wellformed source signatures*: signatures that have identity tags at each module binding and functor body. This is trivially maintained by typing and anchoring. We have the following result:

Theorem 4: Untagging

Given a tag-wellformed signature $\mathtt{S}$, untagging it and tagging it back yields an equivalent signature:

$$\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}' \implies \Gamma \vdash \mathtt{S} \approx [\![\mathtt{S}']\!]$$

The rest of this section is dedicated to the corresponding proofs. This thesis does not contain the proof of Theorem 4.

Proof of Theorem 3

We show the result by mutual induction on the anchoring of signatures and declarations. The induction hypothesis is:

$$\begin{aligned}
&\Gamma \cdot \Delta \,;\, \theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : \theta \ \wedge\ \Gamma \hookrightarrow \theta \ \wedge\ \mathsf{dom}(\theta) = \overline{\alpha} \\
&\qquad \implies \Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}' \ \wedge\ \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\alpha}(\mathsf{args}(\Delta))\right] = \mathcal{C} \\
&\Gamma \cdot \Delta \,;\, \theta \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta \ \wedge\ \mathsf{dom}(\theta) = \overline{\alpha} \ \wedge\ \Gamma \hookrightarrow \theta \\
&\qquad \implies \Gamma \cdot \Delta \vdash \overline{\mathtt{D}} : \lambda\overline{\beta}.\overline{\mathcal{D}}' \ \wedge\ \overline{\mathcal{D}}'\left[\overline{\beta} \mapsto \overline{\alpha}(\mathsf{args}(\Delta))\right] = \overline{\mathcal{D}}
\end{aligned}$$

The proof is by structural induction on the anchoring derivation.

- A-Sig-FctGen: The conclusion of the rule is the signature anchoring judgment

  $$\Gamma \cdot \Delta \,;\, \theta \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \xhookrightarrow{A.Val} () \to \mathtt{S} : \varnothing$$

  Since the domain of the local map is empty, we just have to show

  $$\Gamma \cdot \Delta \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \ (\mathbf{1})$$

  The premise of the rule is $\Gamma \cdot \Delta \cdot ?\overline{\alpha} \,;\, \theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_\,)$. By induction hypothesis, we have $\Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**2**) and $\mathcal{C}'\left[\overline{\beta} \mapsto \overline{\alpha}\right] = \mathcal{C}$, since $\mathsf{args}(?\overline{\alpha})$ is empty, that is $\lambda\overline{\beta}.\mathcal{C}' = \lambda\overline{\alpha}.\mathcal{C}$.

  Then (1) follows by M-Typ-Sig-GenFct applied to (2).

- A-Sig-FctApp: The rule is

$$\frac{\Gamma \cdot \Delta \cdot ?\overline{\alpha}\,;\theta \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathtt{S}_a : \theta_a \ (\mathbf{1}) \qquad \Gamma \cdot \Delta, !\overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \overset{A.Val(Y)}{\hookrightarrow} \mathtt{S} : \theta \ (\mathbf{2})}{\Gamma \cdot \Delta\,;\theta \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \overset{A.Val}{\hookrightarrow} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$

  with $\mathsf{dom}(\theta_a) = \overline{\alpha}$. By IH applied to (1), we have $\Gamma \cdot \Delta \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a$ (**3**) since $\mathsf{args}(\overline{\alpha}, \overline{\alpha})$ is empty. Let $\overline{\gamma}$ be $\mathsf{dom}(\theta)$. By IH applied to (2), we have $\Gamma \cdot \Delta, !\overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**4**) and $\mathcal{C}'\left[\overline{\beta} \mapsto \overline{\gamma}(\mathsf{args}(\Delta), \overline{\alpha})\right] = \mathcal{C}$ (**5**), since $\mathsf{args}(\Delta, !\overline{\alpha})$ is equal to $\mathsf{args}(\Delta), \overline{\alpha}$. By rule M-Typ-Sig-AppFct applied to (3) and (4), we have:

$$\Gamma \cdot \Delta \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]$$

  It remains to check:

$$\left(\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right)\left[\overline{\beta}' \mapsto \overline{\gamma}(\mathsf{args}(\Delta))\right] \equiv \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$$

  which follows from (5) by composition of substitutions, since β' does not occur free in $\mathcal{C}_a$.

- A-Sig-StrPath and A-Sig-StrNone are immediate by induction.

- For A-Decl-Val, A-Decl-Type, we may easily show that $\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}$ implies $\Gamma \vdash \mathtt{u} : \tau$.

  In both cases, the conclusion is of the form $\Gamma\,;\theta \vdash \mathtt{D} \overset{A}{\hookrightarrow} \mathcal{D} : \varnothing$, it suffices to show $\Gamma \cdot \Delta \vdash \mathtt{D} : \mathcal{D}$ (**1**). In the case of A-Decl-Val, the conclusion is $\Gamma\,;\theta \vdash \mathsf{val}\ x : \tau \overset{A}{\hookrightarrow} (\mathtt{val}\ x : \mathtt{u}) : \varnothing$ and the premise is $\Gamma \vdash \mathtt{u} : \tau$, which implies $\Gamma \vdash \mathtt{u} : \tau$. Then, (1) follows by M-Typ-Decl-Val.

  The case of A-Decl-Type is similar.

- A-Decl-Mod, A-Decl-ModType are immediate by induction.

- A-Decl-Anchor: the conclusion of the rule is:

$$\Gamma \cdot \Delta\,;\theta \vdash \mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \overset{A}{\hookrightarrow} \mathsf{type}\ t = A.t : (\varphi \mapsto A.t)$$

  From the premises, we know that $\varphi \in \Delta$ and $\mathsf{args}(\Delta) = \overline{\alpha_1}\,; \ldots \overline{\alpha_n}$. Hence, $(\mathsf{type}\ t = \alpha)[\varphi \mapsto \varphi\,\overline{\alpha_1}\,; \ldots \overline{\alpha_n}]$ is equal to $\mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle$, which is exactly the declaration we started with. By Rule M-Typ-Decl-TypeAbs, we have

$$\Gamma \cdot \Delta \vdash (\mathsf{type}\ t = A.t) : \lambda\alpha.\mathsf{type}\ t = \alpha$$

- A-Decl-Seq: The Rule is:

$$\frac{\Gamma\,;\theta \vdash \mathcal{D}_1 \overset{A}{\hookrightarrow} \mathtt{D}_1 : \theta_1 \qquad \Gamma, A.\mathcal{D}_1\,;\theta \uplus \theta_1 \vdash \overline{\mathcal{D}_2} \overset{A}{\hookrightarrow} \overline{\mathtt{D}_2} : \theta_2}{\Gamma\,;\theta \vdash \mathcal{D}_1, \overline{\mathcal{D}_2} \overset{A}{\hookrightarrow} \mathtt{D}_1, \overline{\mathtt{D}_2} : \theta_1 \uplus \theta_2}\ \text{A-Decl-Seq}$$

  Writing the domain of $\theta_1 \uplus \theta_2$ as $\mathsf{dom}(\theta_1 \uplus \theta_2) = (\overline{\alpha}_1, \overline{\alpha}_2)$, we need to show that there exists $\mathcal{D}'_1$ and $\overline{\mathcal{D}'_2}$ such that:

$$\Gamma \cdot \Delta \vdash \mathtt{D}_1, \overline{\mathtt{D}_2} : \lambda\overline{\alpha}_1, \overline{\alpha}_2.\mathcal{D}'_1, \overline{\mathcal{D}'_2} \qquad (\mathbf{1})$$

$$\mathcal{D}'_1, \overline{\mathcal{D}'_2}\left[\overline{\beta}_1, \overline{\beta}_2 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta)), \overline{\alpha}_2(\mathsf{args}(\Delta))\right] = \mathcal{D}_1, \overline{\mathcal{D}_2} \qquad (\mathbf{2})$$

  We have by induction hypothesis:

$$\Gamma \cdot \Delta \vdash \mathtt{D}_1 : \lambda\overline{\beta}_1.\mathcal{D}'_1 \quad \wedge \quad \mathcal{D}'_1\left[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\right] = \mathcal{D}_1$$

$$\Gamma \cdot \Delta, A.\mathcal{D}_1 \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \quad \wedge \quad \overline{\mathcal{D}'_2}\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta, A.\mathcal{D}_1))\right] = \overline{\mathcal{D}_2}$$

  We introduce $\overline{\mathcal{D}''_2} = \overline{\mathcal{D}'_2}\left[\overline{\alpha}_1(\mathsf{args}(\Delta)) \mapsto \overline{\beta}_1\right]$.

– Proof of (1)
We use the Rule M-Typ-Decl-Seq:

$$\frac{\Gamma \vdash_A \mathtt{D}_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1(\mathbf{3}) \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}_2} : \lambda\overline{\alpha}.\overline{\mathcal{D}_2}(\mathbf{4})}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}_2} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}_2}}\ \text{M-Typ-Decl-Seq}$$

The premise (3) is immediate by induction hypothesis. For the premise (4), we show by induction that:

$$\Gamma \cdot \Delta, A.\mathcal{D}_1 \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \implies \Gamma \cdot \Delta, \overline{\beta}_1, A.\mathcal{D}'_1 \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}''_2}$$

– Proof of (2)
By definition, we have

$$\mathsf{args}(\Delta, A.\mathcal{D}) = \mathsf{args}(\Delta)$$

As the declaration $\mathcal{D}'_1$ is wellformed in an environment that does not contain the $\overline{\beta}_2$, substituting for them does not change anything. Therefore, we have:

$$\begin{aligned}
&\mathcal{D}'_1, \overline{\mathcal{D}''_2}\left[\overline{\beta}_1, \overline{\beta}_2 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta)), \overline{\alpha}_2(\mathsf{args}(\Delta))\right] \\
&= \mathcal{D}'_1\left[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\right], (\overline{\mathcal{D}''_2}\left[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\right])\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta))\right] \\
&= \mathcal{D}_1, \overline{\mathcal{D}'_2}\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta))\right] \\
&= \mathcal{D}_1, \overline{\mathcal{D}'_2}\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta, A.\mathcal{D}))\right] \\
&= \mathcal{D}_1, \overline{\mathcal{D}_2}
\end{aligned}$$

This conclude this case.

This concludes the proof.

Proof of Theorem 2

We first prove that typed signatures are anchorable, then that the result is equivalent to the signature we started with. We start with the following lemma:

Lemma 5 (Skolemization lemma)**.** *Skolemizing a signature does not change its anchoring:*

$$\Gamma, !\overline{\alpha} \cdot ?\overline{\beta}, \Delta\,;\theta \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathtt{S} : \theta \implies \Gamma \cdot ?\overline{\beta'}, !\overline{\alpha}, \Delta\,;\theta \vdash \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \overset{P}{\hookrightarrow} \mathtt{S} : \theta$$

Proof. The key observation is that, from the source syntax, accessing *local* types does take into account the depth of enclosing applicative functors. The two key rules are:

- A-Decl-Anchor: when anchoring a type, the number of (list of) universally quantified parameters n does not appear in the anchoring (nor in the resulting declaration).
- A-Type-Application: similarly, the first type arguments τ_1 to τ_k are ignored. If the anchoring point $\theta(\varphi)$ is parameterized by only $n-k$ arguments, adding another argument in front does not change the anchoring.

□

Coming back to the proof of Theorem 2, we first show that typed signatures are anchorable:

$$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \wedge \Gamma \hookrightarrow \theta \implies \exists \mathtt{S}'.\ \Gamma \cdot ?\overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S}' : \overline{\alpha} \mapsto \_$$

We proceed by induction on the typing derivation of signatures and declarations:

- M-Typ-Sig-ModType and M-Typ-Sig-ModType are base-cases of the induction. We easily show that the anchoring of the environment $\Gamma \hookrightarrow \theta$ implies that stored module-types are anchorable.

- M-Typ-Sig-GenFct and M-Typ-Sig-Str are immediate.

- M-Typ-Sig-AppFct: the induction hypothesis gives us that the signature of the core of the functor is anchorable before the substitution of $\overline{\beta}$ for $\overline{\beta'}(\overline{\alpha})$. Here, we use the skolemization lemma.

- M-Typ-Sig-Trans: we first note that all type variables $\overline{\alpha}$ appear in $\mathcal{C}'$. Then, as subtyping between types is only defined by equality, all the types expressions $\overline{\tau}$ appear in $\mathcal{C}$. As $\mathcal{C}$ is anchorable, so are all its type components, making $\mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]$ anchorable.

Finally, the equivalence between source signatures is an immediate consequence of Theorem 3.

4.4 The Foundations: F^ω Elaboration

The M^ω system is designed to offer a standard, standalone, and expressive approach to the typing of ML modules, while hiding the complexity and artifacts of its encoding in F^ω. Yet, the elaboration of module expressions and signatures of M^ω in F^ω, which we now present, served as a basis for the design of M^ω and still shines a new light on its internal mechanisms. It is also used as a proof of type soundness. This elaboration is largely based on the work of Rossberg et al. [2014], but differs in a key manner for the treatment of abstract types defined inside applicative functors. A main contribution is the introduction of *transparent* existential types, an intermediate between the standard existential types, called *opaque* existential types, and the absence of abstraction. They bring the treatment of applicative and generative functors closer, and significantly simplify the elaboration.

In Section 4.4.1, we introduce F^ω with primitive records, existential types and predicative kind polymorphism. In Section 4.4.2, we present the elaboration of M^ω signatures as F^ω types. In Section 4.4.3, we focus on the key mechanism of *repacking* that allows to extend the scope of an existential type. Repacking is the justification of the lifting of existential types through record types. In Section 4.4.4, we introduce transparent existential types to justify the skolemization of existential types through record types. In Section 4.4.5, we show that transparent existential types can actually be encoded *internally* as an F^ω library. In Section 4.4.6, we present the elaboration of modules as F^ω terms.

4.4.1 F^ω with Kind Polymorphism

We use a variant of explicitly typed F^ω with primitive records (including record concatenation), existential types, and predicative kind polymorphism. While primitive records and existential types are standard, kind polymorphism is less common. Predicativity of kind polymorphism is not needed for type soundness. However, it ensures coherence of types used as a logic, that is, it prevents typing terms with the empty type $\forall(\alpha : \star).\alpha$, whose evaluation would not terminate. For that purpose, kinds are split into two categories: large and small. Polymorphic kinds, which are large, can only be instantiated by small kinds, which in turn do not contain polymorphic kinds. In our setting, kind polymorphism is not essential, as it is only used to internalize the encoding of transparent existential types as F^ω-terms in Section 4.4.5. Alternatively, we could have assumed a family of transparent existential type operators indexed by small kinds, so as to never use large kinds, moving part of the encoding to the meta-level.

$$\varsigma := \star \mid \omega \mid \varsigma \to \varsigma \qquad \text{(small kinds)}$$
$$\kappa := \varsigma \mid \forall \omega.\kappa \mid \kappa \to \kappa \qquad \text{(large kinds)}$$
$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall(\alpha : \kappa).\tau \mid \exists^{\blacktriangledown}(\alpha : \kappa).\tau \mid \lambda(\alpha : \kappa).\tau \mid \tau\,\tau \mid \forall\omega.\tau \mid \Lambda\omega.\tau \mid \tau\,\varsigma \mid () \qquad \text{(types)}$$
$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda(\alpha : \kappa).e \mid e\,\tau \mid \Lambda\omega.e \mid e\,\varsigma \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$
$$\mid \mathsf{pack}\,\langle \tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}(\alpha : \kappa).\tau \mid \mathsf{unpack}\,\langle \alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$
$$\Gamma := \cdot \mid \Gamma, \omega \mid \Gamma, \alpha : \kappa \mid \Gamma, x : \tau \qquad \text{(environments)}$$

Figure 16: Syntax of F^ω

The syntax of F^ω is given in Figure 16. Typing rules are standard and given in Figure 17. Type equivalence, defined by $\alpha\beta$-conversion and reordering of record fields, is also standard and omitted. We use letters τ and e to range over types and expressions to distinguish them from types u and expressions e of the core language, even though these should actually be seen as a subset of τ and e. We consider F^ω to be explicitly typed and explicitly kinded. As a convention, we use a wildcard "_" when a type annotation is unambiguously determined by an immediate sub-expression and may be omitted. This is just a syntactic convenience to avoid redundant type information and improve readability, but the underlying terms should always be understood as explicitly-typed F^ω terms. We write ω for kind variables, α and β for type variables of any kind, and φ and ψ for type variables known to be of higher-order kinds. Application of expressions $e\,\varsigma$ and types $\tau\,\varsigma$ to kinds are restricted to small kinds ς. In expressions and type expressions, we actually write kinds κ (and kind abstraction $\lambda\omega$) in pale color so that they are non intrusive, and we often leave them implicit. We actually always do so in the elaboration typing rules below for conciseness.

For convenience, we use n-ary notations for homogeneous sequences of type-binders. We introduce let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as syntactic sugar for $(\lambda(x : \_).e_2)\,e_1$; we define n-ary pack and unpack operators as follows:

$$\begin{array}{ll}
\mathsf{pack}\,\langle \tau\bar{\tau}, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha\bar{\alpha}.\sigma & \triangleq \mathsf{pack}\langle \tau, \mathsf{pack}\,\langle \bar{\tau}, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\bar{\alpha}.\sigma[\alpha \mapsto \tau]\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha\bar{\alpha}.\sigma \\
\mathsf{unpack}\,\langle \alpha\bar{\alpha}, x\rangle = e_1\,\mathsf{in}\,e_2 & \triangleq \mathsf{unpack}\,\langle \alpha, y\rangle = e_1\,\mathsf{in}\,\mathsf{unpack}\,\langle \bar{\alpha}, x\rangle = y\,\mathsf{in}\,e_2 \\
\mathsf{pack}\,\langle \varnothing, e\rangle\,\mathsf{as}\,\sigma \triangleq e & \qquad \mathsf{unpack}\,\langle \varnothing, x\rangle = e_1\,\mathsf{in}\,e_2 \triangleq \mathsf{let}\ x = e\ \mathsf{in}\ e_2
\end{array}$$

4.4.2 Encoding of Signatures

M^ω signatures are actually F^ω types with some syntactic sugar. In F^ω, we see Y and A_I as term variables, similar to x's. We assume a collection ℓ_I of record labels indexed by identifiers I of the source language. Structural signatures $\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}$ are just syntactic sugar for record types $\{\overline{\mathcal{D}}\}$. A small trick is needed to represent type fields, which have no computational content, but cannot be erased during elaboration as they carry additional typing constraints. We reuse the solution of *F-ing* (Rossberg et al. [2014]), encoding them as identity functions with type annotations. For this, we introduce the following syntactic sugar for the term representing a type field (on the left). We overload the notation to also mean its type (on the right).

$$\langle\!\langle \tau : \kappa \rangle\!\rangle \triangleq \Lambda(\varphi : \kappa \to \star).\lambda(x : \varphi\,\tau).x \quad \text{(Term)} \qquad \langle\!\langle \tau : \kappa \rangle\!\rangle \triangleq \forall(\varphi : \kappa \to \star).\varphi\,\tau \to \varphi\,\tau \quad \text{(Type)}$$

The type τ is used as argument of a higher-kinded type operator φ to uniformly handle the encoding of types of any kind. The key (and only useful) property is that two types (of the same kind) are equal if and only if their encodings are equal. Finally, declarations are

$$\vdash \cdot \qquad \frac{\vdash \Gamma \qquad \omega \notin \Gamma}{\vdash \Gamma, \omega} \qquad \frac{\Gamma \vdash \kappa \qquad \alpha \notin \Gamma}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \star \qquad x \notin \Gamma}{\vdash \Gamma, x : \tau}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star} \qquad \frac{\vdash \Gamma \qquad \omega \in \Gamma}{\Gamma \vdash \omega} \qquad \frac{\Gamma \vdash \kappa \qquad \Gamma \vdash \kappa'}{\Gamma \vdash \kappa \to \kappa'} \qquad \frac{\Gamma, \omega \vdash \kappa}{\Gamma \vdash \forall \omega . \kappa}$$

(a) Environment checking

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star} \qquad \frac{\overline{\Gamma \vdash \tau : \star} \qquad \vdash \Gamma}{\Gamma \vdash \{\overline{\ell_l : \tau}\} : \star} \qquad \frac{\vdash \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall (\alpha : \kappa) . \tau : \star}$$

$$\frac{\Gamma, \omega \vdash \tau : \star}{\Gamma \vdash \forall \omega . \tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \exists^{\blacktriangledown} (\alpha : \kappa) . \tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \kappa'}{\Gamma \vdash \Lambda (\alpha : \kappa) . \tau : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1 \, \tau_2 : \kappa}$$

$$\frac{\Gamma, \omega \vdash \tau : \kappa}{\Gamma \vdash \Lambda \omega . \tau : \forall \omega . \kappa} \qquad \frac{\Gamma \vdash \tau : \forall \omega . \kappa \qquad \Gamma \vdash \varsigma}{\Gamma \vdash \tau \, \varsigma : \kappa [\omega \mapsto \varsigma]}$$

(b) Type checking

$$\text{F-Var} \ \frac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{F-Abs} \ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda (x : \tau) . e : \tau \to \tau'} \qquad \text{F-App} \ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \, e_2 : \tau}$$

$$\text{F-Record} \ \frac{\overline{\Gamma \vdash e : \tau} \qquad \#(\overline{\ell})}{\Gamma \vdash \{\overline{\ell = e}\} : \{\overline{\ell : \tau}\}} \qquad \text{F-Proj} \ \frac{\Gamma \vdash e : \{\ell : \tau, \overline{\ell_1 : \tau_1}\}}{\Gamma \vdash e . \ell : \tau} \qquad \text{F-Append} \ \frac{\Gamma \vdash e_1 : \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma \vdash e_2 : \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \# \overline{\ell_2}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \{\overline{\ell_1 : \tau_1} . \overline{\ell_2 : \tau_2}\}}$$

$$\text{F-Tapp} \ \frac{\Gamma \vdash e : \forall (\alpha : \kappa) . \sigma \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e \, \tau : \sigma [\tau \mapsto \alpha]} \qquad \text{F-Kapp} \ \frac{\Gamma \vdash e : \forall \omega . \tau \qquad \Gamma \vdash \varsigma}{\Gamma \vdash e \, \varsigma : \tau [\omega \mapsto \varsigma]} \qquad \text{F-Tabs} \ \frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \lambda (\alpha : \kappa) . e : \forall (\alpha : \kappa) . \tau}$$

$$\text{F-Kabs} \ \frac{\Gamma, \omega \vdash e : \tau}{\Gamma \vdash \Lambda \omega . e : \forall \omega . \tau} \qquad \text{F-Pack} \ \frac{\Gamma \vdash \exists^{\blacktriangledown} (\alpha : \kappa) . \sigma : \star \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \sigma [\tau \mapsto \alpha]}{\Gamma \vdash \mathsf{pack} \, \langle \tau, e \rangle \, \mathsf{as} \, \exists^{\blacktriangledown} (\alpha : \kappa) . \sigma : \exists^{\blacktriangledown} (\alpha : \kappa) . \sigma}$$

$$\text{F-Unpack} \ \frac{\Gamma \vdash e_1 : \exists^{\blacktriangledown} (\alpha : \kappa) . \tau \qquad \Gamma, \alpha : \kappa, x : \tau \vdash e_2 : \sigma \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \mathsf{unpack} \, \langle \alpha, x \rangle = e_1 \, \mathsf{in} \, e_2 : \sigma}$$

(c) Expression typing

Figure 17: Typing rules of F^ω

syntactic sugar for record entries (distinguished by the category of the identifier):

$$\begin{array}{lcllcl} \mathsf{val}\ x : \tau & \triangleq & \ell_x : \tau & \mathsf{module}\ X : \mathcal{C} & \triangleq & \ell_X : \mathcal{C} \\ \mathsf{type}\ t = \tau & \triangleq & \ell_t : \langle\!\langle \tau \rangle\!\rangle & \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} & \triangleq & \ell_T : \langle\!\langle \lambda\overline{\alpha}.\mathcal{C} \rangle\!\rangle \end{array}$$

4.4.3 Sharing Existential Types by Repacking

The encoding of module expressions as F^ω terms is slightly more involved than for signatures. Although structures and functors are simply encoded as records and functions, a difficulty arises from the need to *lift* existential types to extend their scope, as explained in Section 4.1.2.

Let us first consider the easier generative case. The only construct for handling a term with an abstract type is *unpack*, which allows using the term in a *subexpression*, hence with a limited scope, but not to make an abstract type accessible to the *rest of the program*. Yet, abstract type declarations inside modules have an *open* scope and are visible in the rest of the program. At a technical level, the difficulty comes from the representation of structures. To model them, one needs ordered records (also known as telescopes), where each component can introduce new abstract types accessible to the rest of the record, while standard F^ω only provides non-dependent records.

This observation was at the core of the design of *open existential types* Montagu [2010] and of *recursive type generativity* Dreyer [2007a]. Here, in order to stay in plain F^ω, we reuse and adapt the trick of *F-ing* (Rossberg et al. [2014]): structures are built field by field with a special *repacking* pattern: abstract types are unpacked, shared, but abstractly, with the rest of the structure, and then repacked. This allows the terms to mimic the existential lifting done in the types.

To capture this lifting of existentials out of records, we first introduce a combined syntactic form $\mathsf{repack}^\blacktriangledown\ \langle\bar\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which allows the abstract types of e_1 to appear in the type of e_2[3]:

$$\mathsf{repack}^\blacktriangledown\ \langle\bar\alpha, x\rangle = e_1\ \mathsf{in}\ e_2 \ \triangleq\ \mathsf{unpack}\ \langle\bar\alpha, x\rangle = e_1\ \mathsf{in}\ \mathsf{pack}\ \langle\bar\alpha, e_2\rangle\ \mathsf{as}\ \exists^\blacktriangledown\bar\alpha.\ \_$$

Then, we use it to define a new construct to concatenate two records e_1 and e_2 with disjoint domains, but where e_2 might access the first record, via the bound name x_1, and reuse its abstract types, via the bound variables $\overline{\alpha}$:

$$\mathsf{lift}^\blacktriangledown\langle\bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle \ \triangleq\ \mathsf{repack}^\blacktriangledown\ \langle\bar\alpha, x_1\rangle = e_1\ \mathsf{in}\ \mathsf{repack}^\blacktriangledown\ \left\langle\bar\beta, x_2\right\rangle = e_2\ \mathsf{in}\ x_1 \mathbin{@} x_2$$

It is better understood by the following derived typing rule and its use in the following example.

$$\frac{\Gamma \vdash e_1 : \exists^\blacktriangledown\bar\alpha.\ \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma, \bar\alpha, x_1 : \{\overline{\ell_1 : \tau_1}\} \vdash e_2 : \exists^\blacktriangledown\bar\beta.\ \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \# \overline{\ell_2}}{\Gamma \vdash \mathsf{lift}\langle\bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle : \exists^\blacktriangledown\bar\alpha, \bar\beta.\ \{\overline{\ell_1 : \tau_1}; \overline{\ell_2 : \tau_2}\}}$$

Example 4.4.1. *A simple module M with three type fields, on the left-hand side. The raw encoding (after reduction of administrative let-bindings) on the right-hand side shows how abstract types are shared between components via lifting.*

Source

```
1 module M = struct_A
2   type t = A.t
3   type u = A.u
4   type v = A.t × A.u
5 end
```

Encoding of e

$$\begin{aligned} e = \mathit{lift}^\blacktriangledown\ \langle \alpha,\ x_1 = \mathit{pack}\ \langle (), \{\ell_t = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathit{as}\ \exists^\blacktriangledown\alpha.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\} \\ @\ \mathit{lift}^\blacktriangledown\ \langle \beta,\ x_2 = \mathit{pack}\ \langle (), \{\ell_u = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathit{as}\ \exists^\blacktriangledown\beta.\ \{\ell_u : \langle\!\langle \beta \rangle\!\rangle\} \\ @\ \{\ell_v = \langle\!\langle \alpha \times \beta \rangle\!\rangle\}\rangle \end{aligned}$$

Signature of e

$$\mathcal{C} = \exists\alpha, \beta.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\ ;\ \ell_u : \langle\!\langle \beta \rangle\!\rangle\ ;\ \ell_v : \langle\!\langle \alpha \times \beta \rangle\!\rangle\}$$

[3] We leave the type implicit since the type of repacking is fully determined by the combination of $\bar\alpha$ and the type of e_2

4.4.4 Transparent Existential Types and Their Lifting Through Function Types

The repacking pattern allows lifting existential types outside of record types. Unfortunately, this is insufficient for the applicative case, which uses skolemization to further lift abstract types out of the functor body to the front of the functor. This lifting of existential types though universal quantifiers by skolemization and through arrow types, as done in M^ω, is not definable in F^ω. More precisely, lifting through arrow types is *unsound*, while lifting through universal types is not expressible.

One solution is to avoid skolemization by *a-priori abstraction* over all possible type and term variables, i.e., the whole typing context. Doing so, existential types are always introduced at the front and need not be skolemized. This is the solution followed by the authors of *F-ing* (Rossberg et al. [2014]) and by Shan [2004]. While this suffices to prove soundness, the encoding is impractical for manual use of the pattern – as it requires frequently abstracting over the whole environment – and therefore does not provide a good intuition of what modules really are. The encoding could be slightly improved by abstracting over fewer variables, without really solving the problem of a-priori abstraction.

We instead retain skolemization, following the intuition of the M^ω system, but we tweak the definition of existential types to make their lifting though universal types definable. Namely, we introduce *transparent existential types*, written $\exists^{\nabla\tau}(\alpha : \kappa).\sigma$ to described types that behave as usual existentials $\exists^{\blacktriangledown}(\alpha : \kappa).\sigma$ but remembering the witness type τ for the abstract type α.

We create a transparent existential type with the expression $\mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}(\alpha : \kappa).\sigma$, which behaves much as $\mathsf{pack}\ \langle\tau, e\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha : \kappa).\sigma$, except that the witness type τ remains visible in the result type. A transparent existential type is thus weaker than a usual abstract type, as we still see the witness type. It is still abstract, as α cannot be turned back into its witness type τ and has to be treated abstractly. Two transparent existential types with different witnesses are incompatible. This could be seen as a weakness of transparent existentials, but it is actually a key to their lifting through arrow types.

Transparent existential types do not replace usual existential types, which we here call *opaque* existential types, but come in addition to them. Indeed, an expression of a transparent existential type can be further abstracted to become opaque, using the expression $\mathsf{seal}\ e$, which behaves as the identity but turns the expression e of type $\exists^{\nabla\tau}(\alpha : \kappa).\sigma$ into one of type $\exists^{\blacktriangledown}(\alpha : \kappa).\sigma$.

Transparent existential types may also be used abstractly, with the expression $\mathsf{repack}^{\nabla}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which is the analog of the expression $\mathsf{repack}^{\blacktriangledown}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ but when e_1 is a transparent existential type $\exists^{\nabla\tau}(\alpha : \kappa).\sigma_1$. In both cases, e_2 is typed in a context extended with the abstract types $\overline{\alpha}$ and a variable x of type σ_1. Crucially, e_2 cannot see the witnesses $\overline{\tau}$. However, the abstract type variables $\overline{\alpha}$ may still appear in the type σ_2 of the expression e_2, and therefore it is made transparent again in the result type of $\mathsf{repack}^{\nabla}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which is $\exists^{\nabla\tau}(\alpha : \kappa).\sigma_2$. We do not need a primitive transparent version $\mathsf{unpack}^{\nabla}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, since it can be defined as syntactic sugar for $\mathsf{unpack}\ \langle\alpha, x\rangle = \mathsf{seal}\ e_1\ \mathsf{in}\ e_2$.

Lifting through an arrow type So far, one may wonder what is the advantage of transparent existentials by comparison with opaque existentials. Their key advantage is that we can provide two key additional constructs for lifting transparent existentials across arrow types and universal types – the only reason to have introduced them in the first place. The lifting across an arrow type, written $\mathsf{lift}^{\rightarrow} e$, turns an expression of type $\sigma_1 \to \exists^{\nabla\tau}(\alpha : \kappa).\sigma_2$ into one of type $\exists^{\nabla\tau}(\alpha : \kappa).(\sigma_1 \to \sigma_2)$ as long as α is fresh for σ_1. Since we can observe the witness τ, we can ensure that the choice of the witness does not depend on the value (of type σ_1), allowing us to lift it outside of the function. While this operation seems easy, it crucially depends on existential types begin transparent – this transformation would be unsound with opaque existentials.

$$\textsc{F-Hide}\ \frac{\Gamma \vdash \exists^{\nabla\tau}(\alpha:\kappa).\sigma : \star \qquad \Gamma \vdash e : \sigma[\alpha \mapsto \tau]}{\Gamma \vdash \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}(\alpha:\kappa).\sigma : \exists^{\nabla\tau}(\alpha:\kappa).\sigma}$$

$$\textsc{F-Seal}\ \frac{\Gamma \vdash e : \exists^{\nabla\tau}(\alpha:\kappa).\sigma}{\Gamma \vdash \mathsf{seal}\ e : \exists^{\blacktriangledown}(\alpha:\kappa).\sigma}$$

$$\textsc{F-Hidden}\ \frac{\Gamma \vdash e_1 : \exists^{\nabla\tau}(\alpha:\kappa).\sigma \qquad \Gamma, \alpha:\kappa, x:\sigma \vdash e_2 : \sigma'}{\Gamma \vdash \mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}(\alpha:\kappa).\sigma'}$$

$$\textsc{F-LiftArr}\ \frac{\Gamma \vdash e : \sigma_1 \to \exists^{\nabla\tau}(\alpha:\kappa).\sigma_2}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\nabla\tau}(\alpha:\kappa).\sigma_1 \to \sigma_2}$$

$$\textsc{F-LiftAll}\ \frac{\Gamma \vdash e : \forall(\beta:\kappa').\exists^{\nabla\tau}(\alpha:\kappa).\sigma}{\Gamma \vdash \mathsf{lift}^{\forall} e : \exists^{\nabla\lambda(\beta:\kappa').\tau}(\alpha':\kappa' \to \kappa).\forall(\beta:\kappa').\sigma[\alpha \mapsto \alpha'\,\beta]}$$

Figure 18: Typing rules for transparent existential types

Unsoundness example For instance, let us consider the following example (adapted from the example of Dreyer et al. [2003]). We first define the type of a small record containing a value and a function that can be applied to the value:

$$\varphi \triangleq \lambda\alpha.\{\ell_x : \alpha, \ell_f : \alpha \to \mathsf{unit}\}$$

Then we consider the following expression:

$$\begin{aligned} m \triangleq \lambda x.\quad & \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pack}\ \langle \mathsf{int}, \{\ell_x = 42, \ell_f = \lambda x.()\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\varphi\,\alpha) \\ & \mathsf{else}\ (\mathsf{pack}\ \langle \mathsf{unit}\to\mathsf{unit}, \{\ell_x = \lambda u.(), \ell_f = \lambda x.x\,()\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\varphi\,\alpha) \end{aligned}$$

It has type $\mathtt{bool} \to \exists^{\blacktriangledown}\alpha.\ \{\ell_x : \alpha, \ell_f : \alpha \to \mathsf{unit}\}$, but it would be unsound to consider it at the type $\exists^{\blacktriangledown}\alpha.\mathtt{bool} \to \{\ell_x : \alpha, \ell_f : \alpha \to \mathsf{unit}\}$. Indeed, assuming m has such type, the following well-typed code would get stuck on the application $42\,()$:

$$\begin{aligned} \mathsf{unpack}\ \langle \alpha, M\rangle = m\ \mathsf{in}\ & \mathsf{let}\ M_1 = M\,\mathsf{true}\ \mathsf{in} \\ & \mathsf{let}\ M_2 = M\,\mathsf{false}\ \mathsf{in} \\ & M_2.f\ (M_1.x) \end{aligned}$$

The takeaway is that a function of type $\sigma_1 \to \exists^{\blacktriangledown}\alpha.\sigma_2$ can return an existential value whose witness type depends on its argument; it would thus be impossible to have a unique witness type to be used for all applications of the function, as required in a function of type $\exists^{\blacktriangledown}\alpha.(\sigma_1 \to \sigma_2)$. The language F^ω does not have dependent types, but has *existential types with dynamically chosen witnesses* (the opaque ones).

Lifting through a universal type Similarly, lifting across a universal type variable β of kind κ', written $\mathsf{lift}^{\forall} e$, turns an expression of type $\Lambda(\beta:\kappa').\exists^{\nabla\tau}(\alpha:\kappa).\sigma$ into one of type $\exists^{\nabla\lambda(\beta:\kappa').\tau}(\alpha':\kappa' \to \kappa).\forall(\beta:\kappa').\sigma[\alpha \mapsto \alpha'\,\beta]$, provided β is fresh for τ, using skolemization of both the existential variable α and its witness type τ. It is a key for type soundness that β does not appear free in the witness type, hence that we *know* the witness type – this is why skolemization is not encodable with opaque existential types.

An extension of F^ω To summarize, we have extended the syntax of F^ω as follows:

$$\begin{aligned} \tau\ &::=\ \ldots \mid \exists^{\nabla\tau}(\alpha:\kappa).\sigma \\ e\ &::=\ \ldots \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}(\alpha:\kappa).\sigma \mid \mathsf{seal}\ e \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2 \mid \mathsf{lift}^{\to} e \mid \mathsf{lift}^{\forall} e \end{aligned}$$

Their typing rules are given in Figure 18 and discussed below. Transparent existential types are introduced (just as opaque ones) by *packing*, by the Rule F-Hide. They can be transformed

into opaque ones by *sealing*, where the witness is lost, in Rule F-Seal.

$$\textsc{F-Hide}\ \frac{\Gamma \vdash \exists^{\nabla\tau}(\alpha:\kappa).\sigma : \star \qquad \Gamma \vdash e : \sigma[\alpha \mapsto \tau]}{\Gamma \vdash \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}(\alpha:\kappa).\sigma : \exists^{\nabla\tau}(\alpha:\kappa).\sigma} \qquad \textsc{F-Seal}\ \frac{\Gamma \vdash e : \exists^{\nabla\tau}(\alpha:\kappa).\sigma}{\Gamma \vdash \mathsf{seal}\ e : \exists^{\blacktriangledown}(\alpha:\kappa).\sigma}$$

Transparent existential types do not have an unpack construct, only a repack construct, with the following rule:

$$\textsc{F-Repack}\ \frac{\Gamma \vdash e_1 : \exists^{\nabla\tau}(\alpha:\kappa).\sigma \qquad \Gamma, \alpha:\kappa, x:\sigma \vdash e_2 : \sigma'}{\Gamma \vdash \mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}(\alpha:\kappa).\sigma'}$$

This comes from the fact that, when unpacking $\mathsf{unpack}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ the right-hand-side expression e_2 only sees the left-hand-side expression e_1 through the variables α and x. Specifically, the result could not be packed back as a transparent existential in e_2, as α is only seen abstractly! Therefore, we only have a repack construct, and the normal unpack can be obtained by sealing and using the normal opaque unpack construct.

Finally, we have two rules for the lifting operators F-LiftArr and F-LiftAll, written here without kinds for the sake of readability (the kinded version can be found in Figure 18):

$$\textsc{F-LiftArr}\ \frac{\Gamma \vdash e : \sigma_1 \to \exists^{\nabla\tau}(\alpha).\sigma_2}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\nabla\tau}(\alpha).\sigma_1 \to \sigma_2} \qquad \textsc{F-LiftAll}\ \frac{\Gamma \vdash e : \forall\beta.\exists^{\nabla\tau}(\alpha).\sigma}{\Gamma \vdash \mathsf{lift}^{\forall} e : \exists^{\nabla\lambda\beta.\,\tau}(\alpha').\forall\beta.\sigma[\alpha \mapsto (\alpha'\ \beta)]}$$

Semantics The added constructs have no additional computational content, namely repack behaves as a let-binding, while the other constructs behave as their underlying expression e.

Derived operators We add syntactic sugar for n-ary versions of transparent packing and repacking, as we did for opaque existentials. We write seal^n for n applications of seal. We can define a lifting operation $\mathsf{lift}^{\nabla}\langle \bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle$ for dependent record concatenation as the counterpart of the opaque version, by replacing opaque repacking by transparent repacking. Finally, we also define a new operation $\mathsf{lift}^* e$ that uses a combination of the primitive $\mathsf{lift}^{\to}$ and $\mathsf{lift}^{\forall}$ to turn an expression e of type $\forall\bar\alpha.\sigma_1 \to \exists^{\nabla\overline{\tau}}(\bar\beta).\sigma_2$ into one of type $\exists^{\nabla\overline{\lambda\bar\alpha.\,\tau}}(\bar\beta').\forall\alpha.\sigma_1 \to \sigma_2\left[\bar\beta \mapsto \overline{\beta'\,\bar\alpha}\right]$, which is the key transformation for lifting existentials out of applicative-functor bodies. The operator lift^* is defined as $\mathsf{lift}_q^{\forall^p\triangleright}$ where p and q represent the size[4] of $\bar\alpha$ and $\bar\beta$, and $\mathsf{lift}_q^{\forall^p\triangleright}$ is itself inductively defined as follows:

$$\mathsf{lift}^{\diamond}_{q+1} e \triangleq \mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = \mathsf{lift}^{\forall^p}(\mathsf{lift}^{\triangleright} e)\ \mathsf{in}\ \mathsf{lift}^{\diamond}_q x \qquad \mathsf{lift}^{\diamond}_0 e \triangleq e$$

$$\mathsf{lift}^{\forall^{p+1}} e \triangleq \mathsf{lift}^{\forall}(\Lambda\alpha.\mathsf{lift}^{\forall^p}(e\,\alpha)) \qquad \mathsf{lift}^{\forall^0} e \triangleq e$$

- We allow $\mathsf{lift}^{\forall}$ to cross a sequence of quantifiers defining $\mathsf{lift}^{\forall(q+1)} e$ to be $\mathsf{lift}^{\forall}\left(\lambda\alpha.\mathsf{lift}^{\forall q} e\,\alpha\right)$ and $\mathsf{lift}^{\forall 0} e$ to be e.
- We allow $\mathsf{lift}^{\to}$ and $\mathsf{lift}^{\forall q}$ to be applied to a sequence of quantifiers $\exists^{\blacktriangledown}(\bar\alpha:\overline{\omega}).\tau$ instead of a single one, defining $\mathsf{lift}^{\diamond}_{n+1} e$ to be $\mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = \mathsf{lift}^{\diamond} e\ \mathsf{in}\ \mathsf{lift}^{\diamond}_n x$ with $\mathsf{lift}^{\diamond}_0 e$ equal to e.
- We define $\mathsf{lift}_n^{\forall q\to} e$ to be $\mathsf{lift}_n^{\forall q}(\mathsf{lift}_n^{\to} e)$.

We then simply write $\mathsf{lift}^* e$ for $\mathsf{lift}_n^{\forall q\to} e$, leaving the number of iterations n and q implicit from the type of the argument (taking the longest possible sequence). We have the following derived typing rule:

$$\textsc{F-LiftStar}\ \frac{\Gamma \vdash e : \forall\alpha.\sigma \to \exists^{\nabla\overline{\tau}}(\beta).\sigma'}{\Gamma \vdash \mathsf{lift}^* e : \exists^{\nabla\overline{\lambda\alpha.\,\tau}}(\beta').\forall\alpha.\sigma \to \sigma'[\beta \mapsto (\beta'\ \overline{\alpha})]}$$

[4] p and q are left implicit in $\mathsf{lift}^* e$ as they can be determined from the type of the argument e

4.4.5 Implementation of Transparent Existential Types in F^ω

Interestingly, transparent existential types are completely definable in plain F^ω (with kind polymorphism). A concrete implementation is given in Figure 19, with and without syntactic sugar.

We first define a record e_0 with the constructs, which is actually pretty simple: most fields are η-expansions of the identity. Then, we define a type expression $\tau_{\mathbb{E}}$, and use normal (opaque) existentials of F^ω to pack e_0: $e_{\mathbb{E}} = \mathsf{pack}\,\langle \tau_0, e_0\rangle\ \mathsf{as}\ \tau_{\mathbb{E}}$ where τ_0 is the interface type that hides the implementation of the type $\mathbb{E}$. Using this definition, we may see a program e using transparent existential types as a program $\mathsf{unpack}\ \langle \mathbb{E}, x_{\mathbb{E}}\rangle = e_{\mathbb{E}}\ \mathsf{in}\ e$ in plain F^ω, with the following additional syntactic sugar[5]:

$$
\begin{array}{rcl}
\exists^{\triangledown\tau}(\beta:\kappa).\sigma & \triangleq & \mathbb{E}\,\kappa\,\tau\,(\lambda(\beta:\kappa).\sigma) \\
\mathsf{pack}\ e\ \mathsf{as}\ \exists^{\triangledown\tau}(\alpha).\sigma & \triangleq & x_{\mathbb{E}}.\mathsf{Pack}\,\tau\,(\lambda(\alpha:\_).\sigma)\,e \\
\mathsf{repack}^{\triangledown}\,\langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2 & \triangleq & x_{\mathbb{E}}.\mathsf{Repack}\ \_\ \_\ \_\ (\Lambda(\alpha:\_).\lambda(x:\alpha).e_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{seal}\ e & \triangleq & x_{\mathbb{E}}.\mathsf{Seal}\ \_\ \_\ e \\
\mathsf{lift}^{\rightarrow} e & \triangleq & x_{\mathbb{E}}.\mathsf{Lift}^{\rightarrow}\ \_\ \_\ e \\
\mathsf{lift}^{\forall}\ e & \triangleq & x_{\mathbb{E}}.\mathsf{Lift}^{\forall}\ \_\ \_\ e
\end{array}
$$

We also write $\mathsf{repack}^{\diamondsuit}\,\langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ and $\mathsf{lift}^{\diamondsuit}\langle \bar{\alpha}, x_1 = e_1\ @\ e_2\rangle$ where $\diamondsuit$ stands for either $\triangledown$ or $\blacktriangledown$.

Sound by construction The key take-away of this implementation is that it inherits soundness from F^ω: we do not need to do a soundness proof of the extended language. We verified this implementation in an F^ω prototype and in Coq via a shallow embedding.

4.4.6 Elaboration

As for M^ω, the elaboration relies on a subtyping judgment and a typing judgment for both signatures and modules. However, as M^ω signatures are already F^ω types, we can reuse the M^ω elaboration judgment (although we should now reread it with implicit kinds). Specifically, neither M^ω signatures nor its typing contexts mention transparent existential types. This is a key observation: transparent existential types may only appear in types of module expressions. This means that values of such types are never bound to a variable (during elaboration), which would otherwise force them to appear in the typing context. Instead, transparent existential types are always lifted to the top of the expression (using the lift operators).

There are two main elaboration judgments, for subtyping and typing.

- $\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit}\overline{\alpha}.\mathcal{C} \rightsquigarrow e$ the typing of the module expressions and the declarations. In addition to the signature, an *evidence term* e is produced. The judgment is also defined for bindings $\Gamma \vdash_A \mathtt{D} : \exists^{\diamondsuit}\overline{\alpha}.\mathcal{D} \rightsquigarrow e$
- $\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f$ the subtyping between M^ω signatures, extended to return the explicit coercion function f. The judgment is also defined for declarations $\Gamma \vdash \mathcal{D} \prec: \mathcal{D}' \rightsquigarrow f$

Properties of those two judgments are stated and proved in the next section, Theorem 7.

Subtyping

Subtyping rules are given in Figure 21 and discussed below.

[5]Here, _ stands for kinds or types that are left implicit as they can be straightforwardly inferred from other arguments. We also extend transparent existentials with sequences of abstractions as we did for opaque existentials.

$$
\begin{array}{lcl}
e_0 & \triangleq & \left\{\begin{array}{ll}
\mathsf{Pack} & = \Lambda\omega.\,\Lambda(\alpha:\omega).\,\Lambda(\varphi:\omega\to\star).\,\lambda(x:\varphi\,\alpha).\,x \\
\mathsf{Seal} & = \Lambda\omega.\,\Lambda(\alpha:\omega).\,\Lambda(\varphi:\omega\to\star).\,\lambda(x:\varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha,x\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}(\alpha:\omega).\,\varphi\,\alpha \\
\mathsf{Repack} & = \Lambda\omega.\,\Lambda(\alpha:\omega).\,\Lambda(\varphi:\omega\to\star).\,\lambda(x:\mathbb{E}\,\omega\,\alpha\,\varphi). \\
 & \qquad \Lambda(\psi:\omega\to\star).\,\lambda(f:\forall(\alpha:\omega).\,\varphi\,\alpha\to\psi\,\alpha).\,(f\,\alpha\,x) \\
\mathsf{Lift}^{\to} & = \Lambda\omega.\,\Lambda(\alpha:\omega).\,\Lambda(\varphi:\omega\to\star).\,\Lambda(\beta:\star).\,\lambda(f:\beta\to\varphi\,\alpha).\,f \\
\mathsf{Lift}^{\forall} & = \Lambda\omega.\,\Lambda\omega.\,\Lambda(\alpha:\omega\to\omega).\,\Lambda(\varphi:\omega\to\omega\to\star).\,\lambda(x:(\forall(\beta:\omega).\,\varphi\,\beta(\alpha\,\beta))).\,x
\end{array}\right. \\
\tau_0 & \triangleq & \Lambda\omega.\,\lambda(\alpha:\omega).\,\lambda(\varphi:\omega\to\star).\,\varphi\,\alpha \\
\tau_{\mathbb{E}} & \triangleq & \exists^{\blacktriangledown}(\mathbb{E}:\forall\omega.\,\omega\to(\omega\to\star)\to\star). \\
 & & \left\{\begin{array}{ll}
\mathsf{Pack} & :\ \forall\omega.\,\forall(\alpha:\omega).\,\forall(\varphi:\omega\to\star).\,\varphi\,\alpha\to\mathbb{E}\,\omega\,\alpha\,\varphi \\
\mathsf{Seal} & :\ \forall\omega.\,\forall(\alpha:\omega).\,\forall(\varphi:\omega\to\star).\,\mathbb{E}\,\omega\,\alpha\,\varphi\to\exists^{\blacktriangledown}(\alpha:\omega).\,\varphi\,\alpha \\
\mathsf{Repack} & :\ \forall\omega.\,\forall(\alpha:\omega).\,\forall(\varphi:\omega\to\star).\,\mathbb{E}\,\omega\,\alpha\,\varphi\to \\
 & \qquad \forall(\psi:\omega\to\star).\,(\forall(\alpha:\omega).\,\varphi\,\alpha\to\psi\,\alpha)\to\mathbb{E}\,\omega\,\alpha\,\psi \\
\mathsf{Lift}^{\to} & :\ \forall\omega.\,\forall(\alpha:\omega).\,\forall(\varphi:\omega\to\star).\,\forall(\beta:\star). \\
 & \qquad (\beta\to\mathbb{E}\,\omega\,\alpha\,\varphi)\to\mathbb{E}\,\omega\,\alpha\,(\lambda(\alpha:\omega).\,\beta\to\varphi\,\alpha) \\
\mathsf{Lift}^{\forall} & :\ \forall\omega.\,\forall\omega.\,\forall(\alpha:\omega\to\omega).\,\forall(\varphi:\omega\to\omega\to\star). \\
 & \qquad (\forall(\beta:\omega).\,\mathbb{E}\,\omega\,(\alpha\,\beta)\,(\varphi\,\beta))\to \\
 & \qquad \mathbb{E}\,(\omega\to\omega)\,\alpha\,(\lambda(\alpha:\omega\to\omega).\,\forall(\beta:\omega).\,\varphi\,\beta\,(\alpha\,\beta))
\end{array}\right. \\
e_{\mathbb{E}} & \triangleq & \mathsf{pack}\,\langle\tau_0,e_0\rangle\,\mathsf{as}\,\tau_{\mathbb{E}}
\end{array}
$$

(a) Implementation of transparent existentials as a library in F^ω with (predicative) kind polymorphism. The (kind-polymorphic) operator $\mathbb{E}$ should be understood as the transparent existential quantifier.

$$
\begin{array}{lcl}
e_0 & \triangleq & \left\{\begin{array}{ll}
\mathsf{Pack} & = \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\varphi\,\alpha).\,x \\
\mathsf{Seal} & = \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha,x\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\
\mathsf{Repack} & = \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\mathbb{E}\,\alpha\,\varphi).\,\Lambda\psi.\,\lambda(f:\forall\alpha.\,\varphi\,\alpha\to\psi\,\alpha).\,(f\,\alpha\,x) \\
\mathsf{Lift}^{\to} & = \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f:\beta\to\varphi\,\alpha).\,f \\
\mathsf{Lift}^{\forall} & = \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:(\forall\beta.\,\varphi\,\beta(\alpha\,\beta))).\,x
\end{array}\right. \\
\tau_0 & \triangleq & \lambda\alpha.\,\lambda\,\varphi\,.\,\varphi\,\alpha \\
\tau_{\mathbb{E}} & \triangleq & \exists^{\blacktriangledown}(\exists^{\triangledown}.). \\
 & & \left\{\begin{array}{ll}
\mathsf{Pack} & :\ \forall\alpha.\,\forall\,\varphi\,.\,\varphi\,\alpha\to\exists^{\triangledown\alpha}\beta.\,\varphi_\beta \\
\mathsf{Seal} & :\ \forall\alpha.\,\forall\,\varphi\,.\,\exists^{\triangledown\alpha}\beta.\,\varphi_\beta\to\exists^{\blacktriangledown}\beta.\,\varphi_\beta \\
\mathsf{Repack} & :\ \forall\alpha.\,\forall\,\varphi\,.\,\exists^{\triangledown\alpha}\beta.\,\varphi_\beta\to\forall\,\psi\,.\,(\forall\alpha.\,\varphi\,\alpha\to\psi\,\alpha)\to\exists^{\triangledown\alpha}\beta.\,\psi_\beta \\
\mathsf{Lift}^{\to} & :\ \forall\alpha.\,\forall\,\varphi\,.\,\forall\beta.\,(\beta\to\exists^{\triangledown\alpha}\gamma.\,\varphi_\gamma)\to\exists^{\triangledown\alpha}\gamma.\,(\beta\to\varphi\,\gamma)_\gamma \\
\mathsf{Lift}^{\forall} & :\ \forall\alpha.\,\forall\,\varphi\,.\,(\forall\beta.\,\exists^{\triangledown\alpha\beta}\gamma.\,(\varphi\,\beta)_\gamma)\to\exists^{\triangledown\alpha}\gamma.\,(\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta))_\gamma
\end{array}\right. \\
e_{\mathbb{E}} & \triangleq & \mathsf{pack}\,\langle\tau_0,e_0\rangle\,\mathsf{as}\,\tau_{\mathbb{E}}
\end{array}
$$

(b) Same implementation as above, but with omitted kinds and syntactic sugar for the transparent existential operator. We write a type variable as subscript as a shortcut to indicate that it is the free type variable (via an η-expension): φ_β for $\lambda\beta.\,(\varphi\,\beta)$.

Figure 19: Implementation of transparent existentials as a library in F^ω (with and without syntactic sugar)

```
Notation "⋆" := Type.

Definition Pack (ϰ : ⋆) (α : ϰ) (φ : ϰ →⋆) (x : φ α) := x.
Definition Seal (ϰ : ⋆) (α : ϰ) (φ : ϰ →⋆) (x : φ α) := existT φ α x.
Definition Repack (ϰ : ⋆) (α : ϰ) (φ : ϰ →⋆) (x : φ α)
  (ψ : ϰ →⋆) (f : forall β, φ β →ψ β) := f α x.
Definition Lift_ar (ϰ : ⋆) (α : ϰ) (φ : ϰ →⋆)
  (β : ⋆) (f : β →φ α) := f.
Definition Lift_for (ω ϰ : ⋆) (α : ω →ϰ) (φ : ω →ϰ →⋆)
  (x : forall β : ω, φ β (α β)) := x.

Record τ (𝔼 : forall ϰ, ϰ →(ϰ →⋆) →⋆) := mkTau
  { pack : forall ϰ (α : ϰ) (φ : ϰ →⋆), φ α →𝔼 ϰ α φ;
    seal : forall ϰ (α : ϰ) (φ : ϰ →⋆), 𝔼 ϰ α φ →{α:ϰ & φ α};
    repack : forall ϰ (α : ϰ) (φ : ϰ →⋆), 𝔼 ϰ α φ →
       forall (ψ : ϰ →⋆), (forall β:ϰ, φ β →ψ β) →𝔼 ϰ α ψ;
    lift_ar : forall ϰ (α : ϰ) (φ : ϰ →⋆) (B : ⋆),
          (B →𝔼 ϰ α φ) →𝔼 ϰ α (fun β ⇒B →φ β);
    lift_for : forall ω ϰ (α : ω →ϰ) (φ : ω →ϰ →⋆),
          (forall b:ω, 𝔼 ϰ (α b) (φ b)) →
          𝔼 (ω →ϰ) α (fun (α : ω →ϰ) ⇒forall (β:ω), φ β (α β)) }.

Definition τ0 ϰ (α : ϰ) φ : ⋆ := φ α.
Definition e0 := mkTau τ0  Pack Seal Repack Lift_ar Lift_for.
Definition e𝔼 := existT τ τ0 e0.
```

Figure 20: Implementation of transparent existentials in Coq

Declarations As subtyping between declarations is restricted to the equality for type and value fields, the coercion functions of rules E-Sub-Decl-Val and E-Sub-Decl-Type are just the identity:

E-Sub-Decl-Val
$$\Gamma \vdash (\mathsf{val}\ x : \tau) \prec: (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Type
$$\Gamma \vdash (\mathsf{type}\ t = \tau) \prec: (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$$

For submodules, the coercion function is just returned directly:

E-Sub-Decl-Mod
$$\frac{\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \prec: (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

Finally, for module-types, the coercion function is again the identity:

E-Sub-Decl-ModType
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \prec: \mathcal{C} \rightsquigarrow g}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec: (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

Signatures The rules for signature display coercion functions that are a bit more involved. First, for structural signatures, the coercion function of each field is gathered separately and used to produce the result:

E-Sub-Sig-Struct
$$\frac{\overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}}_0 \prec: \overline{\mathcal{D}}' \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}}')}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec: \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end} \rightsquigarrow \lambda x.\ \left\{\overline{\ell_{I'} = f\,(x.\ell_{I'})}\right\}}$$

For applicative functors, we have two coercion functions: f for the domain and g for the codomain:

E-Sub-Sig-AppFct
$$\frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \prec: \mathcal{C}' \rightsquigarrow g}{\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \prec: \forall\overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}' \rightsquigarrow \lambda x : (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}).\, \Lambda\overline{\alpha}'.\, \lambda(y : \mathcal{C}'_a).\, g\,(x\,\overline{\tau}\,(f\,y))}$$

$$\textsc{E-Sub-Sig-Struct}\quad \frac{\overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}}_0 \prec: \overline{\mathcal{D}}' \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}}')}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec: \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end} \rightsquigarrow \lambda x.\ \left\{\overline{\ell_{I'} = f\,(x.\ell_{I'})}\right\}}$$

$$\textsc{E-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda u.\,\mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}$$

$$\textsc{E-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \prec: \mathcal{C}' \rightsquigarrow g}{\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \prec: \forall\overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}' \rightsquigarrow \lambda x : (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}).\Lambda\overline{\alpha}'.\lambda y : \mathcal{C}'_a.\, g\,(x\,\overline{\tau}\,(f\ y))}$$

(a) Subtyping between signatures

$$\textsc{E-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) \prec: (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x \qquad \textsc{E-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) \prec: (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$$

$$\textsc{E-Sub-Decl-Mod}\quad \frac{\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \prec: (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

$$\textsc{E-Sub-Decl-ModType}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \prec: \mathcal{C} \rightsquigarrow g}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec: (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

(b) Subtyping between declarations

Figure 21: Subtyping with elaboration (outputs the coercion function)

The resulting coercion function takes a function x of type $\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$ as input, then a set of type parameters $\overline{\alpha}'$, and finally a parameter y of type $\mathcal{C}'_a$. To call x, we first specialize it to the types $\overline{\tau}$, then coerce y so it has the type $\mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]$ by using f. Therefore, we have $x\ \overline{\tau}\ (f\ y)$, which is of type $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. This is itself coerced back into $\mathcal{C}'$ by calling g. Overall, it can be thought of as being more or less the composition $g \circ (x\ \overline{\tau}) \circ f$

For generative functors, we have a single coercion function f for the codomain, but we also need to take care of the existential quantification on both sides, that might differ:

$$\textsc{E-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda u.\,\mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}$$

That is, the coercion function repacks the result of the functor with the right type, using the coercion function of the codomain.

Typing

To factor notations for the typing judgment, we introduce the meta-variable ϑ that stands for either an opaque existential $\blacktriangledown$ or a transparent one $\triangledown\tau$ together with its witness type τ. We write $\mathsf{mode}(\vartheta)$ (*resp.* $\mathsf{mode}(\bar{\vartheta})$) for the mode of ϑ (*resp.* the homogeneous sequence $\bar{\vartheta}$), which is either $\triangledown$ or $\blacktriangledown$. When a mode is expected without a witness type, we may leave the projection implicit and just write $\bar{\vartheta}$ instead of $\mathsf{mode}(\bar{\vartheta})$. The convention is the same as for the M^ω system.

The judgment $\Gamma \vdash^{\diamondsuit} \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e$ extends M^ω typing with the elaborated module term e. The mode $\diamondsuit$ must coincide with $\bar{\vartheta}$, and may be left implicit, as we did for the corresponding M^ω

judgment. Hence, we usually just write $\Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e$. A similar, helper judgment $\Gamma \vdash^{\diamondsuit}_{A} \mathtt{B} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{D} \rightsquigarrow e$ is also defined for bindings. When reading an M^ω type environment Γ in F^ω, we must read $A.(\mathsf{val}\ x : \tau)$ and $A.(\mathsf{module}\ X : \mathcal{C})$ as $A_x : \tau$ and $A_X : \mathcal{C}$, etc.

The typing rules are given in Figure 22 and discussed below.

Sequences and structures The key rule for structures is the sequence rule that combines bindings. It may be concisely written as follows for generative and applicative modes:

E-Typ-Bind-SeqGen

$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\blacktriangledown}\overline{\alpha}_1.\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\blacktriangledown}\overline{\alpha}_2.\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\blacktriangledown}\overline{\alpha}_1\overline{\alpha}_2.(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\blacktriangledown}\langle\overline{\alpha}_1, x_1 = e_1 \mathbin{@} (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

E-Typ-Bind-SeqApp

$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\nabla\overline{\tau}_1}(\overline{\alpha}_1).\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\nabla\overline{\tau}_2}(\overline{\alpha}_2).\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}; \overline{\mathtt{B}} : \exists^{\nabla\overline{\tau}_1\overline{\tau}_2}(\overline{\alpha}_1\overline{\alpha}_2).(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\nabla}\langle\overline{\alpha}_1, x_1 = e_1 \mathbin{@} (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

The single field of e_1 is concatenated with the fields of e_2 after lifting out their existential bindings. In both cases, the field of e_1 is made visible in e_2, as well as the existentials in front of e_1—but abstractly. As those two rules are actually similar, they can be factored into a unified rule E-Typ-Bind-Seq:

E-Typ-Bind-Seq

$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\bar\vartheta_1}\overline{\alpha}_1.\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\bar\vartheta_2}\overline{\alpha}_2.\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\bar\vartheta_1\bar\vartheta_2}\overline{\alpha}_1\overline{\alpha}_2.(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\diamondsuit}\langle\overline{\alpha}_1, x_1 = e_1 \mathbin{@} (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

We also have a unified rule for typing structures in both modes:

E-Typ-Mod-Struct

$$\frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\bar\vartheta}\bar\alpha.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e}$$

Modes and sealing By default, elaboration is done in applicative mode, hence inferring transparent existentials, but it can be turned into generative mode when required, using the floating Rule E-Typ-Mod-Seal, which, unsurprisingly, uses the sealing operator:

E-Typ-Mod-Seal

$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}^{|\overline{\alpha}|}\ e}$$

Since signature ascription is defined on paths, it is always applicative (rule E-Typ-Sig-App):

E-Typ-Mod-Ascr

$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma \vdash \overline{\tau} : \overline{\overline{\varsigma}}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow \mathsf{pack}\ f\ e\ \mathsf{as}\ \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C}}$$

That is, signature ascription $(P : \mathtt{S})$ may introduce new abstract types $\overline{\alpha}$ as prescribed by the (elaboration $\lambda\overline{\alpha}.\mathcal{C}$ of the) signature $\mathtt{S}$, but these are transparent existentials in the type of $(P : \mathtt{S})$. The fact that ascription actually does not introduce opaque existential is essential for sealing inside applicative functors, a feature often left out of formalization attempts. Indeed, if ascription were to produce opaque existential, it could not be used inside an applicative functor.

Elaboration of functors At first glance, the elaboration of functors seems to differ quite significantly for the applicative case (Rule E-Typ-Mod-AppFct) and the generative case (Rule E-Typ-Mod-GenFct):

$$\textsc{E-Typ-Mod-Arg}\ \frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \vdash Y : \mathcal{C} \rightsquigarrow Y} \qquad \textsc{E-Typ-Mod-Var}\ \frac{(A.X : \ \mathsf{module}\ \mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C} \rightsquigarrow A_X}$$

$$\textsc{E-Typ-Mod-AppFct}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\nabla\overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*(\Lambda\overline{\alpha}.\,\lambda(Y : \mathcal{C}_a).\,e)}$$

$$\textsc{E-Typ-Mod-AppApp}\ \frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \rightsquigarrow e \qquad \Gamma \vdash P' : \mathcal{C}' \rightsquigarrow e' \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e\,\overline{\tau}\ (f\,e')}$$

$$\textsc{E-Typ-Mod-GenFct}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).e} \qquad \textsc{E-Typ-Mod-GenApp}\ \frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e\,()}$$

$$\textsc{E-Typ-Mod-Ascr}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma \vdash \overline{\tau} : \overline{\varsigma}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow \mathsf{pack}\ f\ e\ \mathsf{as}\ \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C}}$$

$$\textsc{E-Typ-Mod-Seal}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}^{|\overline{\alpha}|}\ e}$$

$$\textsc{E-Typ-Mod-Struct}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e}$$

$$\textsc{E-Typ-Mod-Proj}\ \frac{\Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad (\mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}) \subseteq \overline{\alpha}'}{\Gamma \vdash \mathtt{M}.X : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{clean}^{\diamondsuit}\langle\overline{\alpha}, \overline{\alpha}'\rangle\ (\mathsf{repack}^{\diamondsuit}\langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ x.\ell_X)}$$

(a) Rules for modules

$$\textsc{E-Typ-Bind-Let}\ \frac{\Gamma \vdash \mathtt{e} : \tau \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{let}\ x = \mathtt{e}) : (\mathsf{val}\ x : \tau) \rightsquigarrow \{\ell_x = e\}} \qquad \textsc{E-Typ-Bind-Type}\ \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\ t = \mathtt{u}) : (\mathsf{type}\ t = \tau) \rightsquigarrow \{\ell_t = \langle\!\langle\tau\rangle\!\rangle\}}$$

$$\textsc{E-Typ-Bind-ModType}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ \ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \rightsquigarrow \{\ell_T = \langle\!\langle\lambda\overline{\alpha}.\mathcal{C}\rangle\!\rangle\}} \qquad \textsc{E-Typ-Bind-Empty}\ \ \Gamma \vdash_A \varnothing : \varnothing \rightsquigarrow \{\}$$

$$\textsc{E-Typ-Bind-Mod}\ \frac{\Gamma \vdash_A \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{module}\ X = \mathtt{M}) : (\exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{module}\ X : \mathcal{C}) \rightsquigarrow \mathsf{repack}^{\diamondsuit}\langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \{\ell_X = x\}}$$

$$\textsc{E-Typ-Bind-Seq}\ \frac{\Gamma \vdash_A \mathtt{B} : \exists^{\bar{\vartheta}_1}\overline{\alpha}_1.\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\bar{\vartheta}_2}\overline{\alpha}_2.\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\bar{\vartheta}_1\bar{\vartheta}_2}\overline{\alpha}_1\overline{\alpha}_2.(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\diamondsuit}\langle\overline{\alpha}_1, x_1 = e_1\ @\ (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

(b) Rules for bindings

Figure 22: Typing rules with elaboration

E-Typ-Mod-AppFct
$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\nabla\overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*\Lambda\overline{\alpha}.\,\lambda(Y : \mathcal{C}_a).e}$$

E-Typ-Mod-GenFct
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).e}$$

The body of an applicative functor is elaborated to transparent existentials which are lifted, while in the generative case, the existentials are opaque and cannot be lifted. However, this difference is largely artificial as a result of using a special argument () to enforce generativity. Otherwise, the main difference lies in enforcing the body of the functor to be typed in generative mode, hence with an opaque existential. Since lift^* is neutral on terms that do not have transparent existential types, the elaboration of the generative case could also be written $\mathsf{lift}^*\lambda(\_ : ()).e$, so that the two cases only differ by the modes of elaboration of their bodies.

Functor applications The corresponding rules for applying applicative functors E-Typ-Mod-AppApp and generative functors E-Typ-Mod-GenFct are relatively straightforward, using the normal F^ω application:

E-Typ-Mod-AppApp
$$\frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \rightsquigarrow e \qquad \Gamma \vdash P' : \mathcal{C}' \rightsquigarrow e' \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e\,\overline{\tau}\ (f\,e')}$$

E-Typ-Mod-GenFct
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).e}$$

Projection As expected, projection is elaborated into a record projection, but there is a catch: the projected module expression `M` can have existential types. The solution, displayed in the following simplified rule, is to unpack, project, and pack again, i.e., to repack over the projection:

E-Typ-Mod-Proj-SIMPLIFIED
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \vdash \mathtt{M}.X : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow \mathsf{repack}^\Diamond\,\langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ x.\ell_X}$$

As for structures and sequences, we have a single rule for both modes that relies on the mode of the $\mathsf{repack}^\Diamond$ construct. To exactly match the M^ω rule M-Typ-Mod-Proj for projection, we also need to "garbage-collect" the abstract variables that do not appear in the result type. This is done with a $\mathsf{clean}^\Diamond\langle\overline{\alpha}, \overline{\beta}\rangle\ e$ macro that removes the type variables in $\overline{\alpha}$ that do not appear in $\overline{\beta}$. It is defined as:

$$\begin{aligned}
\mathsf{clean}^\Diamond\langle\gamma\overline{\alpha}, \gamma\overline{\beta}\rangle\ e &\triangleq \mathsf{repack}^\Diamond\,\langle\gamma, x\rangle = e\ \mathsf{in}\ \mathsf{clean}^\Diamond\langle\overline{\alpha}, \overline{\beta}\rangle\ x \\
\mathsf{clean}^\Diamond\langle\gamma\overline{\alpha}, \gamma'\overline{\beta}\rangle\ e &\triangleq \mathsf{unpack}^\Diamond\,\langle\gamma, x\rangle = e\ \mathsf{in}\ \mathsf{clean}^\Diamond\langle\overline{\alpha}, \gamma'\overline{\beta}\rangle\ x \\
\mathsf{clean}^\Diamond\langle\varnothing, \varnothing\rangle\ e &\triangleq e
\end{aligned}$$

Using this macro, we can define the proper projection rule:

E-Typ-Mod-Proj
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad (\mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}) \subseteq \overline{\alpha}'}{\Gamma \vdash \mathtt{M}.X : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow \mathsf{clean}^\Diamond\langle\overline{\alpha}, \overline{\alpha}'\rangle\ (\mathsf{repack}^\Diamond\,\langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ x.\ell_X)}$$

Bindings The elaboration of bindings produces records with a single field. The rules for value fields, type fields, and module-type fields are straightforward. Using the type encoding for values, which we recall as:

$$\langle\!\langle (\tau : \omega) \rangle\!\rangle = \lambda(\beta : \omega \to \star).\, \lambda(x : (\beta\, \tau)).\, x$$

we get the following rules:

$$\text{E-Typ-Bind-Let}\quad \frac{\Gamma \vdash \mathtt{e} : \tau \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{let}\, x = \mathtt{e}) : (\mathsf{val}\, x : \tau) \rightsquigarrow \{\ell_x = e\}} \qquad \text{E-Typ-Bind-Type}\quad \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\, t = \tau) \rightsquigarrow \{\ell_t = \langle\!\langle \tau \rangle\!\rangle\}}$$

$$\text{E-Typ-Bind-ModType}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\, T = \lambda\overline{\alpha}.\mathcal{C}) \rightsquigarrow \{\ell_T = \langle\!\langle \lambda\overline{\alpha}.\mathcal{C} \rangle\!\rangle\}}$$

The rule for module bindings displays an extrusion, which, like for the rule of sequences E-Typ-Bind-Seq, uses repacking:

$$\text{E-Typ-Bind-Mod}\quad \frac{\Gamma \vdash_A \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{module}\, X = \mathtt{M}) : (\exists^{\bar\vartheta}\bar\alpha.\, \mathsf{module}\, X : \mathcal{C}) \rightsquigarrow \mathsf{repack}^{\diamond}\, \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ \{\ell_X = x\}}$$

Again, we can merge the two modes within a single rule by using the mode-specific *repacking*.

4.4.7 Properties of elaboration

In this section we state and prove the soundness and correctness of the elaboration of M^ω.

Soundness

We start with a lemma regarding subtyping:

Lemma 6 (Soundness of subtyping)**.** *The coercion functions are well-typed in F^ω. For subtyping between signatures, we have:*

$$\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \implies \Gamma \vdash f : \mathcal{C} \to \mathcal{C}' \tag{4.1}$$

For subtyping between declarations, we have (without the syntactic sugar of declarations):

$$\Gamma \vdash (\ell_I : \tau) \prec: (\ell_I : \tau') \rightsquigarrow f \implies \Gamma \vdash f : \tau \to \tau' \tag{4.2}$$

Proof. We proceed by induction on the typing derivation. The proof cases are immediate applications of the typing rules of F^ω.

- E-Sub-Decl-Val and E-Sub-Decl-Type are immediate (the coercion is the identity). E-Sub-Decl-Mod is immediate by induction hypothesis.
- E-Sub-Decl-ModType: for module-types bindings, we use the fact that the kernel of subtyping is a subset of type equivalence.
- E-Sub-Sig-Struct is immediate by induction hypothesis

- E-Sub-Sig-GenFct: we recall the rule:

  E-Sub-Sig-GenFct
  $$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\, \lambda u.\, \mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}'}$$

  We have $\Gamma, \overline{\alpha} \vdash f : \mathcal{C} \to \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}]$ by induction hypothesis. From there,

  $$\begin{aligned} & \Gamma, (x : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}), \overline{\alpha}, y : \mathcal{C} \vdash (f\ y) : \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau}\right] && \text{(By F-App)} \\ \Longrightarrow\ & \Gamma, (x : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}), \overline{\alpha}, y : \mathcal{C} \vdash \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}' : \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}' && \text{(By F-Pack)} \\ \Longrightarrow\ & \Gamma, (x : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}) \vdash \mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}' : \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}' \\ & && \text{(By F-Unpack)} \end{aligned}$$

  We conclude by two applications of F-Abs.

- E-Sub-Sig-AppFct is similar.

□

We then have the main soundness theorem:

Theorem 7: Soundness

The elaboration of a module expression $\mathtt{M}$ returns a term e that is well-typed in F^ω and ensures that $\mathtt{M}$ is well-typed in M^ω:

$$\begin{aligned} \Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \rightsquigarrow e \ &\Longrightarrow\ \Gamma \vdash e : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \quad \wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar{\vartheta}} \bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \ &\Longrightarrow\ \Gamma \vdash e : \exists^{\bar{\vartheta}} \bar{\alpha}.\left\{\overline{\mathcal{D}}\right\} \ \wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\overline{\mathtt{D}} \end{aligned} \tag{4.3}$$

Proof We proceed by induction on the typing derivation. The cases are mostly immediate applications of F^ω rules.

- Immediate cases: E-Typ-Mod-Arg and E-Typ-Mod-Var are immediate by F-Var. The rules for value, type and module-type bindings (E-Typ-Bind-Let, E-Typ-Bind-Type, E-Typ-Bind-ModType) are also immediate by F-Record.

- E-Typ-Mod-AppFct: we recall the typing rule:

  E-Typ-Mod-AppFct
  $$\frac{\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}_a\ (\mathbf{1}) \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla \overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e\ (\mathbf{2})}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\nabla \overline{\lambda \overline{\alpha}.\tau}}(\overline{\beta'}).\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*(\Lambda \overline{\alpha}.\, \lambda (Y : \mathcal{C}_a).\, e)}$$

  By induction hypothesis on (2), we have:

  $$\Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash e : \exists^{\nabla \overline{\tau}}(\overline{\beta}).\mathcal{C}$$

  By F-Abs and F-Tabs, we have

  $$\Gamma \vdash (\Lambda \overline{\alpha}.\, \lambda (Y : \mathcal{C}_a).\, e) : \forall \overline{\beta}.\mathcal{C}_a \to \exists^{\nabla \overline{\tau}}(\overline{\beta}).\mathcal{C}$$

  We conclude by F-LiftStar.

- E-Typ-Mod-GenFct and E-Typ-Mod-GenApp are immediate by induction hypothesis.

- E-Typ-Mod-AppApp, immediate by induction hypothesis, and using the subtyping lemma.

Theorem 8: Completeness

Well-typed M^ω terms and bindings can always be elaborated:

$$\begin{array}{rcl} \Gamma \vdash \mathtt{M} : \exists^{\diamondsuit} \bar{\alpha}.\mathcal{C} & \implies & \exists e, \bar{\vartheta},\ \Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \rightsquigarrow e \ \wedge\ \mathsf{mode}(\bar{\vartheta}) = \diamondsuit \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\diamondsuit} \bar{\alpha}.\overline{\mathcal{D}} & \implies & \exists e, \bar{\vartheta},\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar{\vartheta}} \bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \ \wedge\ \mathsf{mode}(\bar{\vartheta}) = \diamondsuit \end{array} \tag{4.4}$$

Proof Sketch. Completeness can be easily established as the elaboration rules mimic the M^ω typing rules with no additional constraints on the premises, except for transparent existentials. However, these only appear on the types of elaborated modules as a positive information, which is never restrictive. In particular, a transparent existential type is always used abstractly and pushed in the context after dropping the witness type exactly as an opaque existential type, i.e., as in M^ω. □

4.5 Abstract signatures

In this section we extend M^ω with abstract signatures. We restrict abstract signatures to *simple abstract signatures* (as presented in Section 2.4.3): abstract signatures can only be instantiated by signatures that do not contain abstract module-type bindings. Yet, "*simple*" is a bit misleading: the polymorphism that is provided by simple abstract signatures is not just at the level of types, but also at the structure of types themselves. In the rest of this section, we refer to signatures that are not abstract as *concrete signatures* (they might contain abstract signatures as fields)

In Section 4.5.1, we introduce the key points of the encoding of abstract signatures via examples. In Section 4.5.2, we extend F^ω (and M^ω) with functions and tuples at the kind level to represent the form of polymorphism provided by abstract signatures. In Section 4.5.3, we present the extended typing rules of M^ω.

4.5.1 Key intuitions of abstract signatures

In this subsection we introduce the key intuitions of abstract signatures through examples. The extensions presented here (mostly to the kind system) are formally given in Section 4.5.2.

Ascription Abstract signatures in positive positions are easy: they act as existentially quantified types. Informally, after ascription with the following left-hand side source signature, we would get the right-hand side M^ω signature (using the variable Ψ for an abstract signature variable):

```
sig_A
  module type T = A.T
  module X1 : A.T
  module X2 : A.T
end
```

```
∃Ψ.sig
  module type T = Ψ
  module X1 : Ψ
  module X2 : Ψ
end
```

An important point here is that we do not need existential kinds: once the signature has been abstracted away, the fact that it might have been parametric is not expressible anymore and therefore, does not matter.

Kind polymorphism and *flexroots* Let us consider the following functor:

```
module F = (Y : sig_A module type  T = A.T  module X : A.T end) → Y.X
```

As a first approximation, we could try an M^ω signature of F of the following form:

```
module F : ∀Ψ.
  sig module type T = Ψ module X : Ψ end → Ψ
```

Here, the abstract signature variable Ψ can be instantiated by any wellformed signature, but there is a catch: it cannot by instantiated by a *parametric* signature, like $\lambda\alpha.\mathsf{sig\ type}\ t = \alpha\ \mathsf{end}$ (which is not of kind $\star$). We could fix the signature of F to account for parametric signatures with a single type parameter:

```
module F : ∀(Ψ : ⋆ → ⋆).∀(α : ⋆).
  sig module type T = λβ.(Ψ β)   module X : (Ψ α) end → (Ψ α)
```

Every occurrence of the abstract signature introduces a new type variable (here, only one α as there is only one occurrence of T) that is the corresponding parameter of the occurrence. But what if the signature has several type parameters? The functor should support *any* number of type parameters. We can achieve this by using kind polymorphism[6] $\forall\omega.$, where the kind variable ω can be instantiated to represent any arity. To represent arity at the kind level, we introduce *kind tuples* that range from the empty kind tuple $\varnothing$ to an n-tuple $\varsigma_1 \times \cdots \times \varsigma_n$. This gives us the following kind-polymorphic signature for F (that subsumes the two signatures presented above):

```
module F : ∀ω.∀(Ψ : ω → ⋆).∀(α : ω).
  sig module type T = λ(β : ω).(Ψ β)   module X : (Ψ α) end → (Ψ α)
```

The type variable α contains all the parameterized types of Ψ for the module X. This idea of having one variable to hold all the type components of a signature is inspired by the (similar) *flexroot* concept of Shao [1999]. In our setting, flexroots are not records but mere tuples, and we abstract over the kind of the flexroot. Every occurrence of the abstract signature introduces a corresponding flexroot, as every instance might be parameterized differently.

To illustrate this, we consider the following signature:

```
sig_A   module type  T = A.T   module X_1 : A.T   module X_2 : A.T   end
```

Here, X_1 and X_2 have the same signature, but it might be parameterized by different types: each have its own flexroot. This gives the following signature in M^ω:

```
Λω.λΨ, α_1, α_2.sig  module type T = λα.(Ψ α) module X_1 : (Ψ α_1) module X_2 : (Ψ α_2) end
```

By instantiation and reduction, we can get a signature where X_1 and X_2 have different types:

```
sig
    module type T = λα.sig type t = α end
    module X_1 : sig type t = int end
    module X_2 : sig type t = bool end
end
```

Kind functions However, first-order kind polymorphism is not enough: due to the presence of higher-order functors, we need higher-order kinds to express all possible forms of sharing

[6] We used kind polymorphism already in Section 4.4.5 for the implementation of transparent existentials as a library in F^ω. Then, it was only to have a completely internal presentation, without meta-notations. Here, it is not to circumvent meta-notations, but really to express the right polymorphism.

$$
\begin{aligned}
\aleph &:= \bullet \mid \aleph \Rightarrow \aleph && \text{(meta-kinds)}\\
\varsigma &:= \cdots \mid \varnothing \mid \varsigma \times \cdots \times \varsigma && \text{(small kinds)}\\
\kappa &:= \cdots \mid \lambda(\omega : \aleph).\kappa \mid \kappa\,\kappa && \text{(large kinds)}\\
\tau &:= \cdots \mid \varnothing \mid \tau \times \cdots \times \tau && \text{(types)}\\
\Gamma &:= \cdots \mid (\omega : \aleph) && \text{(environments)}
\end{aligned}
$$

Figure 23: Extension of F^ω with kind functions, kind tuples and meta-kinds.

between an abstract signature in a functor's codomain and the functor's parameter. To see why, let us consider this higher-order functor:

```
module type TA = sig_A module type T = A.T module X : A.T end
module F = (Y : ((Y' : TA) → TA)) → struct_B end
```

The functor F could be applied to a functor that either produces a new abstract signature as output or produces a concrete signature that depends on the parameter signature. Therefore, the following signature would be too weak:

```
module F : ∀ω0.∀(Ψ0 : ω0 → ⋆).∀(α0 : ω0).
           ∀ω1.∀(Ψ1 : ω1 → ⋆).∀(α1 : ω1).
           (sig module type T = λ(β : ω1).(Ψ1 ω1)   module X : (Ψ1 α1) end →
             sig module type T = λ(β : ω0).(Ψ0 ω0)   module X : (Ψ0 α0) end) → sig  end
```

Indeed, with such signature the functor F could not take as input a functor that produces a concrete signature. Here, we need to skolemize the kind variable to cover all possible cases, turning ω_0 into an higher-order kind Ω. Skolemization happens both at the kind and type levels: Ψ_0 and α_0 are turned into kind-polymorphic higher-order types. This gives:

```
module F : ∀Ω.∀(Ψ0 : ∀ω.(ω → ⋆) → ω → (Ω ω) → ⋆).∀(α0 : ∀ω.(ω → ⋆) → ω → (Ω ω)).
           ∀ω .∀(Ψ1 : ω → ⋆).∀(α1 : ω).
           (sig module type T = λ(β : ω).(Ψ1 ω)   module X : (Ψ1 α1) end →
             sig module type T = λ(β : (Ω ω)).Ψ1 ω Ψ1 α1 β
                 module X : (Ψ0 ω Ψ1 α1 (α0 ω Ψ1 α1)) end) → sig  end
```

Skolemization This introduction of higher-order kinds to treat higher-order functors via skolemization strongly resembles Biswas [1995]'s introduction of higher-order types to treat higher-order functors. While his type system does not have higher-order types (nor applicative functors), he used skolemization to represent all the possible ways a functor's codomain might depend on its domain. This effectively introduces polymorphism over higher-order types, while no higher-order types are actually introduced. Similarly, we introduce higher-order kinds to represent all the possible ways the kind of an abstract signature in a functor's codomain might depend on its domain, while we do not have kind-parametric signatures.

4.5.2 Extension of F^ω

The extended syntax of F^ω is summed up in Figure 23. We extended the system in two ways:

- **Kind tuples**: We add n-ary kind tuples, with the empty tuple being written $\varnothing$. At the level of types, we add the corresponding empty-tuple type $\varnothing$, and the n-ary constructor for type tuples. Those type tuples are not the type of tuples of core language! They are not at the base kind $\star$, i.e., no term has a type that has a tuple kind.

$$\Gamma \vdash \star : \bullet \qquad \frac{(\omega : \aleph) \in \Gamma}{\Gamma \vdash \omega : \aleph} \qquad \frac{\Gamma \vdash \kappa : \bullet \qquad \Gamma \vdash \kappa' : \bullet}{\Gamma \vdash \kappa \to \kappa' : \bullet} \qquad \frac{\Gamma, (\omega : \aleph) \vdash \kappa : \bullet}{\Gamma \vdash \forall(\omega : \aleph).\kappa : \bullet}$$

$$\frac{\forall i \in [\![1, n]\!] \,.\, \Gamma \vdash \varsigma_i : \bullet}{\Gamma \vdash \varsigma_1 \times \cdots \times \varsigma_n : \bullet} \qquad \frac{\Gamma, (\omega : \aleph) \vdash \kappa : \aleph'}{\Gamma \vdash \lambda(\omega : \aleph).\kappa : \aleph \Rightarrow \aleph'} \qquad \frac{\Gamma \vdash \kappa : \aleph' \Rightarrow \aleph \qquad \Gamma \vdash \kappa' : \aleph'}{\Gamma \vdash (\kappa\, \kappa') : \aleph}$$

(a) Wellformedness of kinds

$$\frac{\forall i \in [\![1, n]\!] \,.\, \Gamma \vdash \tau_i : \kappa_i}{\Gamma \vdash \tau_1 \times \cdots \times \tau_n : \kappa_1 \times \cdots \times \kappa_n} \qquad \Gamma \vdash \varnothing : \varnothing$$

(b) Extension to type-wellformedness of F^ω (in addition to the rules of Figure 17)

Figure 24: Extension of F^ω wellformedness

- **Kind functions and meta-kinds**: We add kind-level lambda-abstractions and kind applications. In addition, we add an extra layer to "type kind expressions", which we call *meta-kinds* and are denoted with $\aleph$. The base meta-kind is $\bullet$, it is the type of plain kinds. Kind functions have an arrow meta-kind, written $\aleph \Rightarrow \aleph$. It is not the meta-kind of arrow kinds $\kappa \to \kappa'$ (the kind of type operators), which are of the base meta-kind $\bullet$, but of *kind-level* functions.

We add the corresponding typing rules, along with a new judgment of kind-wellformedness, written $\Gamma \vdash \kappa : \aleph$, in Figure 24. The kind-wellformedness rules are very similar to the type-wellformedness rules of F^ω, just pushed one level higher. All the kinds previously considered (the base kind $\star$, kind arrows, and polymorphic kinds) are at the base meta-kind $\bullet$. Only kind functions are at a higher meta-kind. The rule for kind application is standard. There are no rules for wellformedness of meta-kinds, they are always wellformed. We introduce β-reduction at the kind level to reduce kind applications:

$$(\lambda(\omega : \aleph).\kappa)\, \kappa' \rightsquigarrow^\beta \kappa\big[\omega \mapsto \kappa'\big]$$

We consider kinds up to the equivalence defined by the reflexive, symmetric, and transitive closure of β-reduction and renaming of bound kind variables. We extend type-equivalence to inject kind equivalence. The other rules of F^ω typing are unchanged, except for the kind variables that appear with a meta-kind. We preferably use the variable Ψ for abstract-signature variables for the sake of readability, but it should be read as a normal type variable just as α.

Module-type declarations of M^ω For a technical reason that we detail in the next section, we need to change the encoding of the module-type fields of signatures and structures. Until now, signatures were stored in module-type fields as parametric types, following Russo [2004]:

```
1 | module type T = λᾱ.C
```

We switch back to existentially quantified signatures, in the style of Rossberg et al. [2014]:

```
1 | module type T = ∃▾ᾱ.C
```

The two presentations are more-or-less equivalent. The only difference lies in the fact that, with existential signatures, the transformation of the binder from $\exists$ to $\forall$ when the signature

is used for a functor parameter does not not have a logic interpretation. This is a minor technical detail.

4.5.3 Typing rules

Equipped with this extension of the type system, we can present the typing rules for M^{ω}. For the sake of readability, we leave meta-kinds and most kinds implicits. The core change is in the signature elaboration, where we use higher-order kinds to represent all the possible instantiations of a signature. The key mechanisms for kind variables are similar to type variables: variables are introduced, extruded, and skolemized. In addition, there is a specific mechanism of *lowering* that removes kind parametricity in positive position. We discuss the key rules and mechanisms below.

Introduction The only point where kind-parametric abstract module-types are introduced is at an abstract module-type binding:

$$\textsc{M-Typ-Decl-ModTypeAbs}$$
$$\Gamma \vdash_A (\texttt{module type}\ T = A.T) : \Lambda\omega.\,\lambda(\Psi : \omega \to \star).\,\mathsf{module\ type}\ T = \lambda(\alpha : \omega).\,(\Psi\,\alpha)$$

The abstract signature variable Ψ is lifted, while the set of type parameters α is left inside the module-type declaration. Elaboration of signatures (and declarations) is not only a parametric signature but a kind-parametric, type parametric signature of the form $\Lambda\overline{\omega}.\,\lambda\overline{\alpha}.\mathcal{C}$.

Extrusion When combining declarations together in a sequence, the kind variables are merged together in front of the declarations as displayed by the following rule:

$$\textsc{M-Typ-Decl-Seq}$$
$$\frac{\Gamma \vdash_A \mathtt{D}_1 : \Lambda\overline{\omega}_1.\,\lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\omega}_1, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}_2} : \Lambda\overline{\omega}_2.\,\lambda\overline{\alpha}_2.\overline{\mathcal{D}_2}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}_2} : \Lambda\overline{\omega}_1, \overline{\omega}_2.\,\lambda\overline{\alpha}_1, \overline{\alpha}_2.\ \mathcal{D}_1, \overline{\mathcal{D}_2}}$$

Skolemization When elaborating the signature of an applicative functor, we skolemize both the kind variables and the abstract type variables (which contain abstract signature variables) of the functor's body (written here with kinds to explicit the skolemization):

$$\textsc{M-Typ-Sig-AppFct}$$
$$\frac{\Gamma \vdash \mathtt{S}_1 : \Lambda\overline{\omega}_1.\,\lambda\overline{\alpha}_1.\mathcal{C}_1 \qquad \Gamma, \overline{\omega}_1, \overline{\alpha}_1, (Y : \mathcal{C}_1) \vdash \mathtt{S}_2 : \Lambda\overline{\omega}_2.\,\lambda\overline{(\alpha_2 : \kappa_2)}.\mathcal{C}_2}{\Gamma \vdash (Y : \mathtt{S}_1) \to \mathtt{S}_2 : \Lambda\overline{\omega}'_2.\,\lambda\overline{(\alpha'_2 : \forall\omega.\,\kappa_2)}.\forall\overline{\omega}_1.\forall\overline{\alpha}_1.\mathcal{C}_1 \to \left(\mathcal{C}_2\left[\overline{\omega_2 \mapsto (\omega'_2\,\overline{\omega}_1)}\right]\left[\overline{\alpha_2 \mapsto (\alpha'_2\,\overline{\omega}_1\,\overline{\alpha}_1)}\right]\right)}$$

The kind variables coming from the functor's body $\overline{\omega}_2$ are skolemized into $\overline{\omega}'_2$, and each occurrence ω_2 is replaced by $(\omega'_2\,\overline{\omega}_1)$. Abstract type variables coming from the functor's body $\overline{(\alpha_2 : \kappa_2)}$ are turned into kind-polymorphic higher-order types $\overline{(\alpha'_2 : \forall\omega.\,\kappa_2)}$; each occurrence α_2 is replaced by $(\alpha'_2\,\overline{\omega}_1\,\overline{\alpha}_2)$. Finally, we can see that the lambda quantification of the functor domain is turned into a universal quantification.

Lowering A new mechanism specific to kind variables is *lowering*. If we consider the type parameters of a signature that are bound by a lambda binder, they are either turned into universal quantification when the signature is used as the domain of a functor parameter, or into existential quantification when the signature is used for ascription or as the codomain of a generative functor. Kind variables behave differently. While we still want kind-parametricity for functors (as displayed above), we do not want existential kinds when using a kind-parametric signature for ascription or the body of generative functors. We use instead *lowering* to remove

kind parameters, and "lower" them to the base kind $\star$. Let us start with the rule for signature of generative functors:

$$\textsc{M-Typ-Sig-GenFct}\quad \frac{\Gamma \vdash \mathtt{S} : \mathcal{S} \qquad \mathcal{S} = \overline{\Lambda\omega.\Lambda\Psi.\lambda\overline{\alpha}}.\lambda\overline{\beta}.\mathcal{C} \qquad \mathcal{S}^{\downarrow} = \exists^{\blacktriangledown}\overline{(\Psi' : \star)}, \overline{\beta}.\mathcal{S}\ \overline{(\varnothing\ (\lambda(\gamma : \varnothing).\Psi')\ \overline{\varnothing})}\ \overline{\beta}}{\Gamma \vdash () \to \mathtt{S} : \mathcal{S}^{\downarrow}}$$

First, the codomain signature $\mathtt{S}$ is elaborated. We reorder the types and kind variables to group together each kind variable ω with the associated signature variable $(\Psi : \omega \to \star)$ and the flexroots $\overline{\alpha}$. The other type variables $\overline{\beta}$ are left untouched. Then, we introduce a new abstract signature variable $(\Psi' : \star)$ at the base kind for each abstract signature variable Ψ (which might be at a higher kind). Finally, we get the lowered signature $\mathcal{S}^{\downarrow}$ by instantiating, in $\mathcal{S}$:

- each kind variable ω by the empty tuple kind $\varnothing$
- each abstract signature variable Ψ by the constant type function $(\lambda(\gamma : \varnothing).\Psi')$
- each flexroot α by the empty tuple type $\varnothing$; higher-order flexroots are instantiated by constant type functions that return the empty tuple: $\lambda(\_ : \varnothing).\varnothing$

Lowering and subtyping The rule for ascription in module expressions M-Typ-Mod-Ascr also features lowering:

$$\textsc{M-Typ-Mod-Ascr}\quad \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \mathcal{S} \qquad \mathcal{S} = \overline{\Lambda\omega.\Lambda\Psi.\lambda\overline{\alpha}}.\lambda\overline{\beta}.\mathcal{C} \qquad \mathcal{S}^{\downarrow} = \lambda\overline{\Psi'}, \overline{\beta}.\mathcal{S}\ \overline{(\varnothing\ (\lambda(\gamma : \varnothing).\Psi')\ \overline{\varnothing})}\ \overline{\beta} \qquad \Gamma \vdash \mathcal{C} < (\mathcal{S}^{\downarrow}\ (\overline{\exists^{\blacktriangledown}\alpha.\mathcal{C}'})\ \overline{\tau})}{\Gamma \vdash^{\triangledown} (P : \mathtt{S}) : \exists^{\triangledown}\overline{\Psi'}, \overline{\beta}.\ \left(\mathcal{S}^{\downarrow}\ \overline{\Psi'}\ \overline{\beta}\right)}$$

The elaborated signature $\mathcal{S}$ is lowered into $\mathcal{S}^{\downarrow}$ similarly to M-Typ-Sig-GenFct. Then, the subtyping is checked between the signature $\mathcal{C}$ of the path P and the lowered signature, instantiated by a list of concrete types $\overline{\tau}$ and a list of existential signatures $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}'$. The returned signature is the lowered one $\mathcal{S}^{\downarrow}$, with an eta-expansion to turn the lambda binder into an existential one. Subtyping between the lowered signature $\mathcal{S}^{\downarrow}$ rather than the kind-polymorphic signature $\mathtt{S}$ is more restrictive – there are programs that would not typecheck – but we do it to prevent the introduction of existential kinds. To understand why the instantiation with existential signatures works, we show the rule in action on an example.

Example 4.5.1. *We show the Rule M-Typ-Mod-Ascr in action. Let us assume that the path P has the following signature $\mathcal{C}$:*

```
C ≜ sig
      module type T = ∃▼α.sig type t = α end
      module X1 : sig type t = int end
      module X2 : sig type t = bool end
      module F : ∀β.sig type t = β end → sig type t = (β list) end
    end
```

Now, we consider the signature $\mathtt{S}$ *and its elaboration* $\mathcal{S}$*:*

```
S ≜ sig_A
      module type T = A.T
      module X1 : A.T
      module X2 : A.T
      module F : (Y : A.T) → A.T
    end
```

```
S ≜ Λω.λ(Ψ : ω → ⋆).λα1, α2, φ.sig
      module type T = ∃▼α.(Ψ α)
      module X1 : (Ψ α1)
      module X2 : (Ψ α2)
      module F : ∀β.(Ψ β) → (Ψ (φ β))
    end
```

Here, we do not want to instantiate the signature $\mathcal{S}$ directly, as we would get as output a

signature with an existential kind. Instead, we first lower the signature to $\mathcal{S}^{\downarrow}$, defined as:

$$\mathcal{S}^{\downarrow} \triangleq \lambda(\Psi' : \star).\,(\mathcal{S}\ \varnothing\ (\lambda(\gamma : \varnothing).\,\Psi')\ \varnothing\ \varnothing\ (\lambda(\gamma : \varnothing).\,\varnothing))$$

After β-reduction of the empty-tuple type arguments, we obtain the following signature:

```
S↓ ≜ λ(Ψ' : ⋆). sig
        module type T = Ψ'
        module X1 : Ψ'
        module X2 : Ψ'
        module F : Ψ' → Ψ'
      end
```

Finally, before subtyping $\mathcal{C}$ and $\mathcal{S}^{\downarrow}$, we instantiate the latter with $(\exists^{\blacktriangledown}\alpha.\,\mathsf{sig\ type}\ t = \alpha\ \mathsf{end})$. The subtyping is then made field by field. For the module fields, we have:

$$\mathsf{module}\ X_1 : \mathsf{sig\ type}\ t = \texttt{int}\ \mathsf{end}\ <\ \exists^{\blacktriangledown}\alpha.\,\mathsf{module}\ X_1 : \mathsf{sig\ type}\ t = \alpha\ \mathsf{end}$$

$$\mathsf{module}\ X_2 : \mathsf{sig\ type}\ t = \texttt{bool}\ \mathsf{end}\ <\ \exists^{\blacktriangledown}\alpha.\,\mathsf{module}\ X_2 : \mathsf{sig\ type}\ t = \alpha\ \mathsf{end}$$

This is shown easily by instantiating α with `int` *and* `bool` *respectively. For the functor field, we have the following successive subtyping relations:*

$$\begin{aligned} &\mathsf{module}\ F : \forall\beta.\mathsf{sig\ type}\ t = \beta\ \mathsf{end} \to \mathsf{sig\ type}\ t = (\beta\ \mathsf{list})\ \mathsf{end} \\ <\ &\mathsf{module}\ F : \forall\beta.\mathsf{sig\ type}\ t = \beta\ \mathsf{end} \to (\exists^{\blacktriangledown}\alpha.\,\mathsf{sig\ type}\ t = \alpha\ \mathsf{end}) \qquad (1) \\ <\ &\mathsf{module}\ F : (\exists^{\blacktriangledown}\beta.\,\mathsf{sig\ type}\ t = \beta\ \mathsf{end}) \to (\exists^{\blacktriangledown}\alpha.\,\mathsf{sig\ type}\ t = \alpha\ \mathsf{end}) \qquad (2) \end{aligned}$$

The subtyping relation (1) is normal covariance and instantiation, while (2) is justified by the fact the parameter's signature is universally quantified, and that β does not appear free in the codomain:

$$(\forall\alpha.\sigma \to \tau)\ <\ (\exists\alpha.\sigma) \to \tau \qquad (\alpha \notin \mathsf{fv}(\tau))$$

Finally, the result signature is $\mathcal{S}^{\downarrow}$ with a (transparent) existential quantifier:

$$\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown}\Psi'.\,(\mathcal{S}^{\downarrow}\,\Psi')$$

Overall, the instantiation by an existential signature creates a signature where abstract types have been "un-extruded" and "un-skolemized", and this signature is a supertype of $\mathcal{C}$. But those abstract types are hidden away in the resulting signature, abstracted by Ψ'. If "un-extrusion" and "un-skolemization" is not possible, then the subtyping fails and there is a typechecking error. This would happen for instance with the following signature (inside $\mathcal{C}$):

```
module F : sig type t = int end → sig type t = (int list) end
```

The subtyping step (1) would still work, but (2) would fail: it would require the functor to be more polymorphic than it really is!

Functor application The last technical point is the following rule for functor application:

$$\frac{\Gamma \vdash P : \forall\omega'.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]\ [\overline{\omega} \mapsto \overline{\varsigma}]}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]\ [\overline{\omega} \mapsto \overline{\varsigma}]}\ \textsc{M-Typ-Mod-AppApp}$$

Here, we extend the *instantiation* mechanism of subtyping to support both instantiation of kinds and instantiation of types. The instantiation of kinds is guided by the module-type fields and does not pose a problem for decidability.

Discussion We believe this extension of M^{ω} supports the main use-cases for abstract signatures. It adds some complexity, but overall the treatment of kind variables is quite similar to type variables. However, if we were to relax the *simple abstract signatures* criterion and allow

instantiation of abstract signatures by signatures containing abstract module-types, the system would become much more involved. Simple abstract signatures can be seen as the level 1 of predicative instantiation. At level 2, we would already need higher-order meta-kinds and several more quantified variables. We believe that the added complexity is not justified regarding the absence of known use-cases.

4.6 Discussion

4.6.1 Signature artifacts

M^ω signatures are more expressive than source signature, but they may also keep *too much information*, revealing the history of the module operations. This may lead to an inferred signature that is not anchorable, while intuitively providing the same type-sharing information as a simpler, anchorable signature. This typically happens when a type variable has become "unreachable", only appearing in sub-expressions. We have identified two such patterns.

In the following, we use the notation $\mathcal{C}[\alpha, \dots]$ to indicate that α appears freely in $\mathcal{C}$ and the notation $\mathcal{C}[(\varphi\,\alpha), \dots]$ to indicate that α appears only in the subexpression $(\varphi\,\alpha)$. In particular, $\mathcal{C}[\varphi, (\varphi\,\alpha)]$ means that α only appears as an argument of φ in $\mathcal{C}$.

Loss of a type argument. The signature $\exists\,\varphi, \alpha.\ \mathcal{C}[\varphi, (\varphi\,\alpha)]$, could be obtained by exporting a functor (providing φ) along with a type obtained by applying this functor to an argument that has later been hidden. The application $\varphi\,\alpha$ keeps trace that the type was obtained by applying φ to α. However, since the argument α is not accessible, this information became useless. By subtyping, we could safely give the module the simpler signature $\exists\,\varphi, \beta.\ \mathcal{C}[\varphi, \beta]$ cutting the (original) link to the functor , which we can state as a subtyping relationship:

$$\exists\,\varphi, \alpha.\ \mathcal{C}[\varphi, (\varphi\,\alpha)] \ <\ \exists\,\varphi, \beta.\ \mathcal{C}[\varphi, \beta]$$

Anchoring the left-hand side will fail since α cannot be anchored, while anchoring the right-hand might succeed. We do not currently allow this simplification during anchoring, since both signatures are not isomorphic in F^ω as there is no coercion going in the opposite direction.

Loss of a type operator. Similarly, the application of a functor may be exported while the functor itself became unreachable. For instance, with two applications of the same functor, we may have a signature of the form $\exists\,\varphi, \alpha, \beta.\ \mathcal{C}[\alpha, \beta, (\varphi\,\alpha), (\varphi\,\beta)]$, which is a subtype of, but not isomorphic to $\exists \alpha, \beta, \alpha', \beta'.\ \mathcal{C}[\alpha, \beta, \alpha', \beta']$. In the special case where the functor is called only once, the signature would be of the form $\exists\,\varphi, \alpha.\ \mathcal{C}[\alpha, (\varphi\,\alpha)]$, which is actually isomorphic to $\exists \alpha, \alpha'.\ \mathcal{C}[\alpha, \alpha']$.

It would be interesting to explore how the type system of M^ω could capture those forms of simplifications without visible loss of type-sharing.

Chapter 5

ZipML

In this chapter, we present ZipML, an almost fully syntactic type system for ML modules. While sharing the same source language as M^ω, ZipML directly works with the OCaml-style syntactic signatures, only enriching them with the new construct of *zipper signatures*. ZipML implements early, shallow and lazy strengthening, and does not eagerly inlines type and module-type definitions. We say that ZipML uses *almost* fully syntactic signatures, as it lightly extends the signature language but keeps all of its constructs for inference, while M^ω replaced the signature language with M^ω-signatures. ZipML is therefore closer to real-world implementations of module systems, and should integrate more easily with them.

Code inserts To distinguish ZipML signatures from sources ones, we use the following convention: ZipML signatures have an orange background (on the right-hand side), as in:

```
1 module M : sig type t = int end
```

```
1 module M : sig type t = int end
```

Overview In Section 5.1, we discuss the motivation for a syntactic system, as opposed to the elaboration approach of M^ω (Chapter 3). In Section 5.2, we present the new technical device of zipper signatures through examples. In Section 5.3, we give the formal definition of ZipML. In Section 5.4, we introduce *zipper simplification*, a process analogous to anchoring that empties useless declarations in zippers. In Section 5.5, we state the soundness theorem of ZipML. We also give a conjecture regarding the completeness of ZipML against M^ω.

5.1 Motivation and challenges of a syntactic system

In this section, we discuss some downsides of M^ω (and more generally, the *F-ing* (Rossberg et al. [2014]) approach). Then, we present the core challenges of a syntactic approach, and give a high-level intuition on how ZipML solves them.

While appealing, the *F-ing* (Rossberg et al. [2014]) approach actually leads to a significant gap between the user writable "source" signatures and their internal representations in F^ω. Users are either required to think in terms of the elaboration, while still writing types in the surface language or to rely on the syntactic types while the internal representation is hidden away with a reverse translation. Both options are unsatisfactory. In the former case, as the elaboration can be quite involved and polluted with a lot of type variables, which makes internal types hard to keep in the users' mind. In the latter case, the reverse translation (*anchoring*) is actually non trivial: it must undo the extrusion and the skolemization of existential types to rebuild abstract type fields.

Besides, the anchoring raises some challenges:

- The process can fail, because types of F^ω, the target language, are more expressive than signatures of the source language: as a result, some inferred signatures are not expressible in the source signature syntax – and the program must then be rejected. This issue, called *signature avoidance*, discussed in more details below, is a serious problem in all syntactic approaches that has not yet received a good solution. Those cases trigger a specific class of typechecking errors that might be tricky to present for the user, exposing the internal representation of types.

- Printing typechecking errors (whether or not due to signature avoidance), requires the reverse translation to be extended to "invalid" source signatures, that cannot be written by the user himself.

- Even when the typechecking succeeds, the M^ω type-system combined with the anchoring is not *fully syntactic* in the sense of Shao [1999]: if a module expression admits a signature, not all sub-expressions necessarily do so. This might be counter-intuitive for the user, as the simple act of exposing a sub-module might make typechecking fail in nontrivial ways.

In the rest of this section, starting with examples in OCaml, we discuss our design for a new syntactic module system with an OCaml-like syntax, called ZipML. Namely,

- we explain how we can delay and "solve" the *signature avoidance* problem by *zipping away* out-of-scope components;

- we maintain a proper sharing of types, via the so-called *strengthening* operation [Leroy, 1994], but with a new strategy: *early, lazy and shallow*;

- we do not use eager expansion of intermediate types and module-type definitions, so as to print concise interfaces respecting the user's intent – an aspect previously ignored in the whole *F-ing* (Rossberg et al. [2014]) line of work.

The last two mechanisms have also been designed with efficiency of typechecking in mind. Laziness allows for more sharing of signatures, which might drastically affect performance when signatures are very large.

5.1.1 Signature avoidance

At a more abstract level, the signature avoidance problem described in Section 2.3 boils down to the projection out of a dependent record: given a signature `S` of a module `X` that depends on a type `t`, as in:

```
module M = struct type t module X : S end
```

How can we extract `S[t]` and keep it wellformed, while it mentions `t`, that used to be in its local context? Sometimes there exists a principal signature `S'` that is equivalent to `S` while *avoiding* the type `t`, but sometimes there exists none. The key issue here is to model projection as a way to *delete* part of a signature (the **`type t`** declaration). However, one could argue that projection is instead a way to *hide* part of a signature, making it inaccessible to the user but not necessarily deleting it. This is for example the stance taken by Harper and Stone [2000]; Dreyer et al. [2003], as their system is designed to make some components invisible to the user but not to the typechecker. However, such an approach requires a somewhat ad-hoc elaboration with reserved names, something along the lines of:

```
module M : sig
  module HIDDEN : sig type t
  module VISIBLE : S
end
```

This technique became less popular than the more *logically grounded* existentials types. We argue that this original intuition was good, but the technical device to achieve it was too heavy. Intuitively, we would like to *focus* on `S` while keeping the rest of the fields *on the side* if needed. A classical technique in functional programming, *zippers* [HUET, 1997], allows precisely to extend any tree-like type definition with a zipper to focus on certain part of the tree while keeping the rest on the side. We therefore change the meaning of projection to be the *focusing*[1] of a signature unto the projected field, keeping the rest in the *zipper context*, also called *floating fields*. In this simple example, ZipML would return[2]:

```
module M : ⟨type t⟩ S
```

ZipML uses zippers to preserve relevant type information and therefore, delay the resolution of signature avoidance until a source signature is forced by ascription. Zippers also behave gracefully in case of errors, allowing to present users with the exact field names that were lost during typechecking. In most of this paper, we only need a simpler form of zipper where we can drop some information, because we never need to *unfocus*. Namely, we can forget the name of the sub-module we projected on (here, `X`) and all fields that may appear *after* it, as they cannot appear in `S`. In Section 5.4, to give a precise meaning to a signature simplification algorithm where we do need to *unfocus*, we re-introduce actual zippers, which we call *full zippers*.

5.1.2 Strengthening

In path-based systems, accessing a module value in the context needs a special treatment. Usually, one would expect a typing rule of the form:

$$\frac{(X : S) \in \Gamma}{\Gamma \vdash X : S}\ \textsc{Var}$$

Applied to a module system, this would give:

```
module X : sig type t end = ...
module X' = X
```

```
module X : sig type t end
module X': sig type t end
```

But this is unsatisfactory, as duplicating the signature would create two different abstract types `X.t` and `X'.t` that would not be equal! This comes from the fact that type sharing is

[1] Despite what the name may suggest it is not (to the best of my understanding) linked with the *focusing logic* used by Crary [2020] to solve signature avoidance.

[2] Ignoring the self-references

expressed with manifest types [Leroy, 1994], i.e., with explicit equalities between type fields. As a consequence, when accessing a module, its signature must be rewritten, hence copied, to refer to the type fields of the original module. This operation, called *strengthening*, is central to *path-based* systems such as OCaml. We write it S/X. ZipML implements strengthening with three innovations: earliness, shallowness and laziness.

Early strengthening In order to always get a strengthened signature when accessing a value from the context, there are two solutions: either we apply strengthening *early*, when pushing the signature in the context, or *late*, when retrieving the signature from the context. ZipML uses the former, as it allows for a simple context access rule.

Shallow and lazy strengthening Rewriting a whole signature to strengthen it can be costly[3] and hinder performance for very large libraries. To prevent useless rewriting while keeping type sharing, ZipML adopts a *lazy* version of strengthening: we reuse transparent signatures `(= P < S)` to delay the strengthening of `S` by `P`, which is computed lazily when actually accessing or inspecting the signature, and therefore avoids some signature duplication. When strengthening is actually required on a signature, it remains shallow: only the top level equalities are rewritten, while the strengthening of the rest of the signature is delayed.

5.1.3 Lazy expansion of definitions

Module languages allow for both type definitions (`type t = int * float`) and module-type definitions (`module type T = sig .. end`). However, while OCaml retains such definitions internally, as any real-word implementation, M^{ω}, *F-ing* (Rossberg et al. [2014]) and most other previous works handle them by *eager inlining* . Inlining of module-type definitions in signatures might considerably increase their size and make typechecking of large-scale libraries with intensive use of modules quite expensive. It also obscures the programmers intent by inferring large signatures whose names or aliases have been lost. ZipML keeps module-type definitions internally and in inferred signatures, instead of systematically inlining them. Consequently the type system relies on a notion of *normalization* to allow the inlining of definitions on demand.

5.2 An introduction to floating fields

The main novelty of ZipML is the introduction of floating fields as a way to delay and resolve instances of the signature avoidance problem. In this section we give an overview of the mechanism of floating fields, including their interaction with self-references. We first show how they provide additional expressiveness compared to OCaml. Then, we show how several zippers are chained together when doing several projections in a row. Finally, we introduce the mechanism of zipper simplification that allows the removal of useless parts of a zipper context. In the following subsection, we give examples in OCaml code (without self-references in structures).

5.2.1 Expressivity

Zippers improve the expressiveness of sources signatures in two ways: they remove signature avoidance cases (which results in typechecking errors) and they prevent the loss of type-sharing that could be caused by the abstraction heuristic of OCaml (see Section 2.3.2).

[3] See the discussion at https://github.com/ocaml-flambda/flambda-backend/pull/1337

Preventing typechecking errors Let us start by giving a precise instance of the pattern presented in Section 5.1.1:

```
module M = (struct
  type t
  module X = struct let l : t list = [] end
end).X
```

If we were to typecheck this in OCaml, (using **open** or an anonymous functor call to simulate the anonymous projection), we would get an error:

```
The value "l" has no valid type if "M" is hidden.
```

Before the projection, the signature of the whole structure is:

$\texttt{sig}_B\ \texttt{type}\ t = B.t\ \ \texttt{module}\ X : \texttt{sig}_A\ \texttt{val}\ l : B.t\ \texttt{list}\ \texttt{end}\ \texttt{end}$

When projecting on the module field X, we would like to return directly:

$\texttt{sig}_A\ \texttt{val}\ l : B.t\ \texttt{list}\ \texttt{end}$

This signature is however ill-formed as it refers to the name $B.t$ that has been lost. The solution is to put the lost fields in a zipper context, accessible via the self-reference B. Therefore, ZipML would typecheck and return the following signature:

$\texttt{module}\ M : \langle B : \texttt{type}\ t = B.t \rangle\ \texttt{sig}_A\ \texttt{val}\ l : B.t\ \texttt{list}\ \texttt{end}$

The highlighted expression $\langle B : \texttt{type}\ t = B.t \rangle$ is a *zipper context* containing a single *floating field*; it is not a signature, but the content of an out-of-focus signature that is bound to the self-reference B.

Preventing loss of type-sharing Let us look at a situation where a lost type appears in other type definitions, as in:

```
module M = (struct
  type t
  module X = struct type u = t * bool type v = t * bool end
end).X
```

Here, instead of failing, OCaml abstracts the definition of `u` and `v`, not only losing their pair structure, but also forgetting that these are actually equal types.

```
sig type u type v end (* over-abstraction *)
```

We say that this is erroneously solved by *over-abstraction*. While over-abstraction is *safe*, and could be obtained by explicit subtyping, it is *incomplete*. ZipML would return the following signature:

$\langle B : \texttt{type}\ t = B.t \rangle\ \texttt{sig}\ \texttt{type}\ u = B.t \times \texttt{bool}\ \ \texttt{type}\ v = B.t \times \texttt{bool}\ \texttt{end}$

This is a better answer, as no type information has been lost.

5.2.2 Chaining zippers

Let us consider how zippers are handled when there are several projections. We define a module `M` by projection of a deeply nested submodule:

```
module M = (struct
  type t
  module X = struct
    type u = t list
```

```
    module Y = struct type v = t type w = u end
  end
end).X.Y
```

OCaml would output the following signature, where all type equalities have been lost (which is a case of *over-abstraction*):

```
sig type v type w end
```

By contrast, ZipML would infer the following signature $\mathtt{S}_0$:

$$\langle A : \mathtt{type}\ t = A.t\rangle\,\langle B : \mathtt{type}\ u = A.t\ \mathtt{list}\rangle\ \ \mathtt{sig}_C\ \mathtt{type}\ v = A.t\ \ \mathtt{type}\ w = B.u\ \mathtt{end}$$

Let us detail the process. The complete signature of the structure before the projection is:

$$\begin{array}{l}\mathtt{sig}_A\\ \quad \mathtt{type}\ t = A.t\\ \quad \mathtt{module}\ X : \mathtt{sig}_B\\ \quad\quad \mathtt{type}\ u = A.t\ \mathtt{list}\\ \quad\quad \mathtt{module}\ Y : \mathtt{sig}_C\ \mathtt{type}\ v = A.t\ \ \mathtt{type}\ w = B.u\ \mathtt{end}\\ \quad \mathtt{end}\\ \mathtt{end}\end{array}$$

It is of the form $\mathtt{S}_A[\mathtt{S}_B[\mathtt{S}_C]]$, where the inner-most signature $\mathtt{S}_C$ is placed inside two surrounding signatures. For the first projection on field `.X`, we turn the outer signature into a zipper context, which gives us: $\langle \mathtt{S}_A \rangle\, \mathtt{S}_B[\mathtt{S}_C]$. For the second projection on field `.Y` we need to project out of a zipper signature, which is not a structural signature. However, it actually composes well: any operation on a zipped signature correspond to pushing the zipper context in the environment, doing the operation and popping the zipper context back. For our projection, we therefore first push the zipper context in the typing environment, leaving us with $\mathtt{S}_B[\mathtt{S}_C]$. Projecting gives $\langle \mathtt{S}_B \rangle\, \mathtt{S}_C$, and popping the zipper context back again gives: $\langle \mathtt{S}_A \rangle\, (\langle \mathtt{S}_B \rangle\, \mathtt{S}_C)$. Finally, we can merge the two zipper contexts in a single one and we obtain $\langle \mathtt{S}_A ; \mathtt{S}_B \rangle\, \mathtt{S}_C$. Conceptually, we can sum up those steps as:

$$\begin{aligned}\left(\mathtt{S}_A\left[\mathtt{S}_B\left[\mathtt{S}_C\right]\right].X\right).Y &\to \left(\langle \mathtt{S}_A \rangle\, \mathtt{S}_B\left[\mathtt{S}_C\right]\right).Y && \text{(First projection)}\\ &\to \langle \mathtt{S}_A \rangle \left(\mathtt{S}_B\left[\mathtt{S}_C\right].Y\right)\\ &\to \langle \mathtt{S}_A \rangle\, (\langle \mathtt{S}_B \rangle\, \mathtt{S}_C) && \text{(Second projection)}\\ &\to \langle \mathtt{S}_A ; \mathtt{S}_B \rangle\, \mathtt{S}_C\end{aligned}$$

Interestingly, each component of the zipped contexts $\mathtt{S}_A$ and $\mathtt{S}_B$ can be directly accessed from $\mathtt{S}_C$ via their self-reference names, respectively A and B.

5.2.3 Zipper simplification

Let us follow-up on the example. We start from the inferred signature $\mathtt{S}_0$:

$$\langle A : \mathtt{type}\ t = A.t\rangle\,\langle B : \mathtt{type}\ u = A.t\ \mathtt{list}\rangle\ \ \mathtt{sig}_C\ \mathtt{type}\ v = A.t\ \ \mathtt{type}\ w = B.u\ \mathtt{end}$$

While $\mathtt{S}_0$ is correct, ZipML would simplify it to a more "economical" – yet equivalent – signature, where useless floating fields have been removed:

$$\mathtt{sig}_C\ \mathtt{type}\ v = C.v\ \ \mathtt{type}\ w = C.v\ \mathtt{list}\ \mathtt{end}$$

Let us detail this process of *zipper simplification*.

Inlining and garbage-collection Here, we can see that the second zipper context is not essential, as we may *inline* the definition of u in the definition of w, which gives us $\mathtt{S}_1$:

```
⟨A : type t = A.t⟩ ⟨B : type u = A.t list⟩ sig_C type v = A.t type w = A.t list end
```

Then, we see that we have an *unused* floating field, as the name u does not appear anymore in the zipped signature. As zippers are static information, not present at runtime, we can drop the second zipper context, which gives us $\mathtt{S}_2$:

```
⟨A : type t = A.t⟩ ⟨ sig_C type v = A.t type w = A.t list end⟩
```

Reordering Informally the signature $\mathtt{S}_2$ could be seen as the result of a projection of the unzipped signature (using some reserved field $\mathbb{Z}$ for the lost projection path):

```
(sig_A
  type t = A.t
  module ℤ : sig_C type v = A.t type w = A.t list end
end).ℤ
```

In the signature $\mathtt{S}_2$ the type field v is an alias to the floating abstract type $A.t$, which comes first. However, since the field `type` $t = A.t$ is floating, hence absent at runtime, it may as well be moved after the module field $\mathbb{Z}$, provided $\mathtt{v}$ becomes the abstract type definition and t becomes an alias of v:

```
(sig_A
  module ℤ : sig_C type v = C.v type w = C.v list end
  type t = A.ℤ.v
end).ℤ
```

The key is that the two unzipped signatures, before their projection, are subtype of one another, hence equivalent! Floating fields that come after the hole can always be dropped as they are not present at runtime and cannot be referenced from the signature. This finally gives us $\mathtt{S}_3$, the signature returned by ZipML:

```
sig_C type v = C.v type w = C.v list end
```

In this case, we were able to eliminate all floating fields and therefore resolve signature avoidance without using zippers in the resulting signature. Interestingly, having a better heuristic in OCaml would not have been sufficient to obtain this signature: between the two projection, we went through a step where the intermediate signature $\langle \mathtt{S}_A \rangle\ \mathtt{S}_B\ [\mathtt{S}_C]$ is not expressible without zippers.

In general, the simplification may remove some but not all floating fields. This is fine, as we are able to pursue typechecking in presence of floating fields, which may perhaps be dropped later on.

Signature avoidance in normal OCaml workflow Developers usually write implementation file `.ml` together with an interface file `.mli`. When typechecking, if an `.mli` is present, an ascription is made between the content of the `.ml` file, say `M` and the content of the `.mli` file, say `S`. The final signature is therefore the result of `(M : S)`. If it succeeds, the signature is the user-written *source signature* `S`, which never contains floating fields. A normal workflow with ZipML would therefore not exhibit zippers at all in case of success, even if they appeared internally during typechecking. In other cases, we may return an answer with floating fields, giving the user the possibility to remove them via a signature ascription, or use zippers to give a meaningful typechecking error.

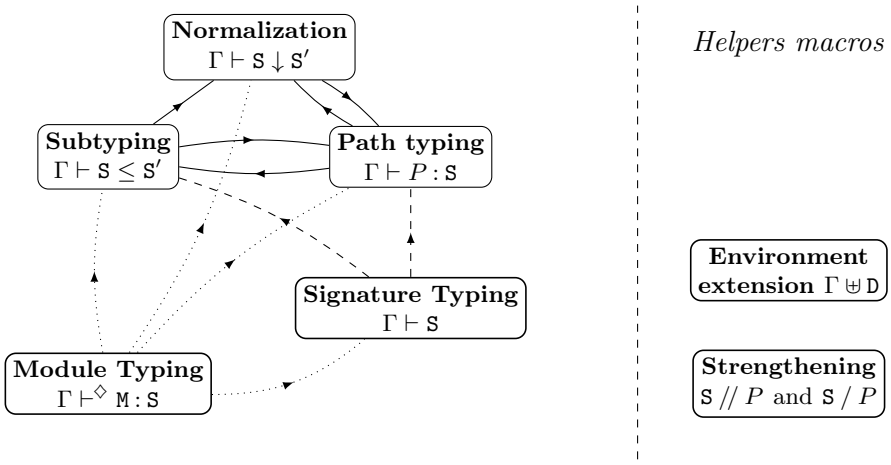


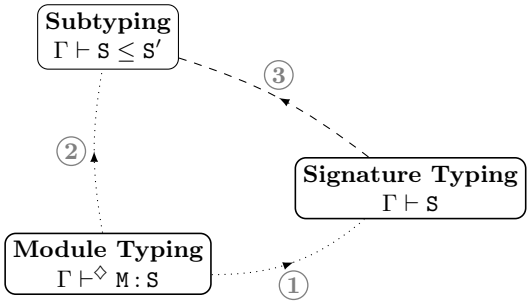
Figure 25: Relationship between the main judgments of ZIPML

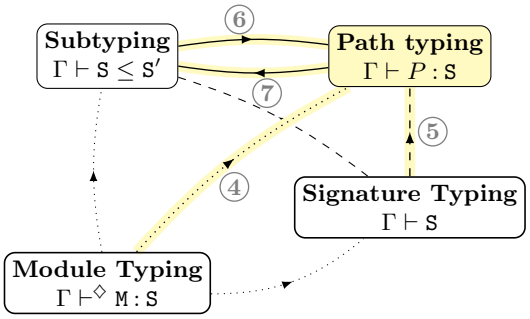
5.3 Formal presentation

5.3.1 Overview

In this section we give an overview of the typing system. The main judgments and their dependencies are summed up in Figure 25. In the following, we break down the structure of the system and role of each judgment.

Core The core functionality of the system is provided by **Module Typing**, **Signature Typing** and **Subtyping**. Module typing is the entry point of the typing system: given a module expression $\mathtt{M}$, it infers its signature $\mathtt{S}$. As users can write signatures in module expressions, we need to check their well-formedness via signature typing, which explains the dependency ①. Signature typing also plays the meta-theoretic role of a wellformedness judgment. As the language features ascriptions (explicit and implicit at functor call), we also need a subtyping judgment. It is used by the module typing, which explains the dependency ②. Finally, ZIPML has transparent signatures, therefore checking a signature may require checking subtyping. This explains the dependency ③.



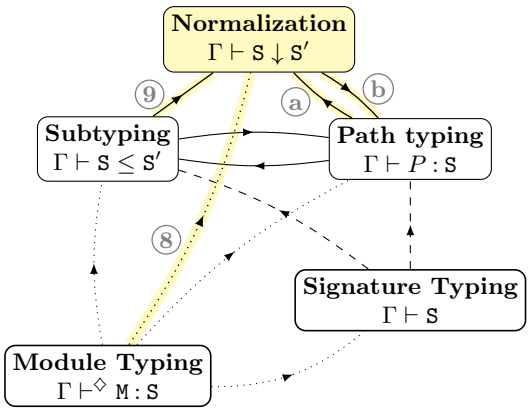


Qualified types and values However, unlike M^{ω}, ZIPML does not introduce abstract type variables, for which wellformedness is easily known, but instead maintains the syntactic paths and *qualified types* in inferred signatures[4]. Therefore, paths appear everywhere in signatures and types. We could consider paths as being just module expressions, as M^{ω} does, but it would make the whole system mutually recursive and more complex. Instead, we extract **Path Typing**

[4]Even with eager inlining of definitions, abstract type fields are qualified types

as a separate judgment[5]. As module expressions contain paths, typing a module expression may require typing a path, which explains the dependency ④. Signature typing and subtyping handle paths, qualified types, and module-types. Therefore, it depend on path typing: this explains the dependencies ⑤ and ⑥. Finally, as the system supports applicative functors, paths may contain functor applications. Typing a functor application requires a subtyping check, which explains the dependency ⑦. In Section 5.3.4, we also define **Path resolution**, written $\Gamma \vdash P \triangleright \mathtt{S}$, as a macro to access the *content* of the module at P.

Lazy inlining and normalization ZipML keeps the user-provided names for types and signatures – unlike M^{ω} which eagerly inlines them. In practice, it means that accessing a component inside a signature, or checking the equality between two types may require inlining of definitions. This is done by the **Normalization** judgment. All judgments that need to actually pattern-match on the head constructor of a signature depend on normalization, which explains the dependencies ⑧, ⑨, and ⓐ. Finally, normalization itself need to follow paths to lookup definitions, which explains the dependency ⓑ. Overall, normalization can (and should) be thought of as a part of the underlying signature equivalence that can be used anywhere in the system. That is, assuming $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$, we have:



$$\Gamma \vdash P : \mathtt{S} \implies \Gamma \vdash P : \mathtt{S}' \qquad \Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S} \implies \Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}'$$

$$\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S} \implies \Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}' \qquad \Gamma \vdash \mathtt{S} \leq \mathtt{S}_2 \implies \Gamma \vdash \mathtt{S}' \leq \mathtt{S}_2$$

Strengthening To complete the picture of Figure 25 we need to add helper judgments that serve as *macros*, as they do not depend on other judgments. We have the following:

- **Delayed strengthening** – $\mathtt{S} /\!/ P$: adds a syntactic mark on $\mathtt{S}$ (a transparent signature) to *delay* the strengthening of $\mathtt{S}$ by P
- **Shallow strengthening** – $\mathtt{S} \,/\, P$: rewrite all top level abstract types to point to their counterpart at P, and uses delayed strengthening on deeper definitions.
- **Environment extension** – $\Gamma \uplus \mathtt{D}$: (lazily) strengthens the right-hand part before pushing it in Γ. It is the key to *early strengthening*.

Plan We start with the grammar extension and technical details in Section 5.3.2. In Section 5.3.3, we present the strengthening helpers. In Section 5.3.4, we define path typing, normalization. In Section 5.3.5, we present the subtyping judgment. In Section 5.3.6, we discuss the signature typing judgment. Finally, in Section 5.3.7, we present module typing.

5.3.2 Grammar extensions

ZipML reuses the syntax of Section 5.3.2, with a few extensions presented in Figure 26: we add zipper signatures and distinguish between signatures with and without zippers.

[5] As paths are always applicative, the path typing judgment does not need a mode annotation, as opposed to module typing

Zipper context

$\gamma ::= \varnothing \mid A : \overline{\mathtt{D}} \mid \gamma \,;\, \gamma$

Signature

$\mathtt{S} ::= \langle \gamma \rangle\, \mathtt{S}$ (Zipper)
$\mid \tilde{\mathtt{S}}$ (Plain signature)
$\tilde{\mathtt{S}} ::= (= P < \tilde{\mathtt{S}})$ (Transparent signature)
$\mid Q.T$ (Module-type)
$\mid () \to \mathtt{S}$ (Generative functor)
$\mid (Y : \tilde{\mathtt{S}}) \to \mathtt{S}$ (Applicative functor)
$\mid \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}$ (Structural signature)

Path and Prefix

$P ::= ...$
$\mid P.A$ (Zipper access)

Typing environment

$\Gamma ::= \varnothing \mid \Gamma, A.I : \mathtt{S} \mid \Gamma, Y : \tilde{\mathtt{S}} \mid \Gamma, \gamma$

Declaration

$\mathtt{D} ::= \mathtt{val}\, x : \mathtt{u}$ (Value)
$\mid \mathtt{type}\, t = \mathtt{u}$ (Type)
$\mid \mathtt{module}\, X : \mathtt{S}$ (Module)
$\mid \mathtt{module\ type}\ T = \tilde{\mathtt{S}}$ (Module-type)

Figure 26: Syntax extension for signatures and typing environments

Typing environments Typing environments Γ bind module fields to declarations $A.I : \mathtt{S}$ and functor parameters to signatures $Y : \tilde{\mathtt{S}}$. In typing environments module fields are always prefixed by a self-reference. As we disallow shadowing, every $A.I$ in Γ must be unique. For convenience, we may write $\Gamma, A.\overline{\mathtt{D}}$ for the sequence $\Gamma, \overline{A.\mathtt{D}}$. Finally, typing environments may also contain floating fields Γ, γ. By associativity, we identify $\Gamma, (A.\overline{\mathtt{D}}, A'.\overline{\mathtt{D}}')$ and $(\Gamma, A, \overline{\mathtt{D}}), A'.\overline{\mathtt{D}}'$.

Zippers We introduce *zipper signatures*, written $\langle \gamma \rangle\, \mathtt{S}$, where γ is the *zipper context* and $\mathtt{S}$ is the *zipped signature*. Zipper contexts are themselves sequences of floating fields $A : \overline{\mathtt{D}}$. In a zipper $\left\langle A : \overline{\mathtt{D}} \right\rangle \mathtt{S}$, the self-reference A, which is used to access fields of $\mathtt{D}$ from $\mathtt{S}$, **cannot** be renamed. The concatenation of zippers contexts $\gamma_1 \,;\, \gamma_2$ is only defined when the *domains* of γ_1 and γ_2 are disjoint, the domain being defined as follows:

$$\mathsf{dom}(\varnothing) = \emptyset \qquad \mathsf{dom}(A : \overline{\mathtt{D}}) = \{A\} \qquad \mathsf{dom}(\gamma_1 \,;\, \gamma_2) = \mathsf{dom}(\gamma_1) \uplus \mathsf{dom}(\gamma_2)$$

We may then see a zipper as a map from self-references to declarations and define $\gamma(A)$ accordingly. The concatenation of zippers ";" is associative and the empty zipper $\varnothing$ is a neutral element. We identify $\mathtt{S}$ and $\langle \varnothing \rangle\, \mathtt{S}$ and $\langle \gamma_1 \rangle\, \langle \gamma_2 \rangle\, \mathtt{S}$ and $\langle \gamma_1 \,;\, \gamma_2 \rangle\, \mathtt{S}$ whenever γ_1 and γ_2 have disjoint domains. Therefore, a signature $\mathtt{S}$ can always be written as $\langle \gamma \rangle\, \tilde{\mathtt{S}}$ where $\tilde{\mathtt{S}}$ is a plain signature and γ concatenates all consecutive zipper contexts or is $\varnothing$ if there are none. The introduction of zippers requires a new form of path $P.A$, invalid in source programs, to access floating fields.[6] Finally, we define plain signatures $\tilde{\mathtt{S}}$ as those without an initial zipper (but where sub-terms may contain zippers). Note that a zipper may not appear under a transparent ascription or a functor parameter.

Invariants We also define several syntactic subcategories of signatures to capture some invariants.[7] The *head value form* $\mathtt{S}^{\mathrm{v}}$ of a signature gives the actual shape of a signature, which is either a structural signature or a functor. The *head normal form* $\mathtt{S}^{\mathrm{n}}$ is similar, but still contains the *identity* of a signature, if it has one, via a transparent ascription.

$$\mathtt{S}^{\mathrm{v}} ::= \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} \mid (Y : \mathtt{S}) \to \mathtt{S} \mid () \to \mathtt{S} \qquad \mathtt{S}^{\mathrm{n}} ::= \mathtt{S}^{\mathrm{v}} \mid (= P < \mathtt{S}^{\mathrm{v}})$$

Notice that head normal forms (and value forms) are superficial and a signature appearing withing value forms may itself be of any form. Hence, it may contain inner zippers.

[6] In the absence of floating fields, self-references could only be at the origin of a path Q.

[7] For sake of readability, we do not always use the most precise syntactic categories and sometimes just write $\mathtt{S}$ when $\mathtt{S}$ may actually be of a more specific form. Conversely, we may use subcategories to restrict the application of a rule that only applies for signatures of a specific shape.

Shallow strengthening

$$\begin{aligned}
\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}\ /\ P &\triangleq \mathtt{sig}_A\ \overline{\mathtt{D}[A \mapsto P]\ /\!/\ P}\ \mathtt{end} \\
(Y : \mathtt{S}_a) \to \mathtt{S}\ /\ P &\triangleq (Y : \mathtt{S}_a) \to (\mathtt{S}\ /\!/\ P(Y)) \\
() \to \mathtt{S}\ /\ P &\triangleq () \to \mathtt{S}
\end{aligned}$$

Delayed strengthening (signatures)

$$\begin{aligned}
(= P' < \mathtt{S})\ /\!/\ P &\triangleq (= P' < \mathtt{S}) \\
\langle \gamma\,;\gamma' \rangle\,\mathtt{S}\ /\!/\ P &\triangleq \big(\langle\gamma\rangle\,\big(\langle\gamma'\rangle\,\mathtt{S}\big)\big)\ /\!/\ P \\
\big\langle A : \overline{\mathtt{D}} \big\rangle\,\mathtt{S}\ /\!/\ P &\triangleq \big\langle A : \overline{\mathtt{D}}[A \mapsto P.A]\ /\!/\ P.A \big\rangle\,\big(\mathtt{S}[A \mapsto P.A]\ /\!/\ P\big) \\
\mathtt{S}\ /\!/\ P &\triangleq (= P < \mathtt{S}) \qquad \text{(Other cases)}
\end{aligned}$$

Delayed strengthening (declarations)

$$\begin{aligned}
(\mathtt{val}\,x : \mathtt{u})\ /\!/\ Q &\triangleq \mathtt{val}\,x : \mathtt{u} \\
(\mathtt{module}\,X : \mathtt{S})\ /\!/\ Q &\triangleq \mathtt{module}\,X : (\mathtt{S}\ /\!/\ Q.X) \\
(\mathtt{type}\,t = \mathtt{u})\ /\!/\ Q &\triangleq \mathtt{type}\,t = \mathtt{u} \\
(\mathtt{module\ type}\ T = \mathtt{S})\ /\!/\ Q &\triangleq \mathtt{module\ type}\ T = \mathtt{S}
\end{aligned}$$

Environment extension

$$\begin{aligned}
\Gamma \uplus (Y : \mathtt{S}) &\triangleq \Gamma, Y : (\mathtt{S}\ /\!/\ Y) \\
\Gamma \uplus (\mathtt{module}\,A.X : \mathtt{S}) &\triangleq \Gamma, \mathtt{module}\,A.X : (\mathtt{S}\ /\!/\ A.X) \\
\Gamma \uplus \big(\big\langle A : \overline{\mathtt{D}} \big\rangle\,\gamma\big) &\triangleq (\Gamma, A : (\overline{\mathtt{D}}\ /\!/\ A)) \uplus \gamma
\end{aligned}$$

Figure 27: Strengthening helpers

To characterize signatures, declarations, and zippers that are fully strengthened, we define a notion of *generalized transparent signature* $\mathbf{S}$ as either a transparent signature or a zipped transparent signature. We use this notion to define generalized transparent declarations and zipper contexts:

$$\begin{aligned}
\mathbf{S} &::= (= P < \tilde{\mathtt{S}}) \mid \langle\boldsymbol{\gamma}\rangle\,(= P < \tilde{\mathtt{S}}) \qquad\qquad \boldsymbol{\gamma} ::= A : \overline{\mathbf{D}} \mid \boldsymbol{\gamma}\,;\boldsymbol{\gamma} \mid \varnothing \\
\mathbf{D} &::= \mathtt{module}\,X : \mathbf{S} \mid \mathtt{val}\,x : \mathtt{u} \mid \mathtt{module\ type}\ T = \mathtt{S} \mid \mathtt{type}\,t = \mathtt{u}
\end{aligned}$$

Finally, we let $\tilde{P}$ stand for *source* paths P that do not contain any (direct or recursive) access to some zipper context (of the form $P'.A$). Consistently, $\tilde{Q}$ means $\tilde{P}$ or A and *source* types $\tilde{\mathtt{u}}$ means types $\mathtt{u}$ where all paths occurring in $\mathtt{u}$ are actually source paths.

5.3.3 Strengthening

The three helper judgments for strengthening are given in Figure 27 and discussed below.

Shallow strengthening is only defined on signatures in head normal form and is used to push the actual strengthening just *one level down*. It rewrites only the top level abstract fields, and delays the deeper ones. This is done by substituting the self-reference by the path:

$$\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}\ /\ P \triangleq \mathtt{sig}_A\ \overline{\mathtt{D}[A \mapsto P]\ /\!/\ P}\ \mathtt{end}$$

Here, all type fields that were of the form $\mathtt{type}\,t = A.t$ get rewritten into $\mathtt{type}\,t = P.t$. Then, delayed strengthening is applied to all the declarations. Strengthening an applicative functor amounts to delaying the strengthening of its body, while there is nothing to do for generative functors (they define a different scope):

$$(Y : \mathtt{S}_a) \to \mathtt{S}\ /\ P \triangleq (Y : \mathtt{S}_a) \to (\mathtt{S}\ /\!/\ P(Y)) \qquad () \to \mathtt{S}\ /\ P \triangleq () \to \mathtt{S}$$

Delayed strengthening The role of delayed strengthening is to insert a transparent ascription at the top level of a signature, if there is not one already, and push it under zippers if any. Transparent signatures already have a delayed strengthening, so we leave them unchanged (similarly to concrete type definitions).

$$(= P' < \mathtt{S}) \mathbin{//} P \triangleq (= P' < \mathtt{S})$$

For zippers, we have two definitions: one to handle sequences of zippers contexts (left-hand side) and one to lazily strengthen a single zipper context (right-hand side):

$$\langle \gamma ; \gamma' \rangle \, \mathtt{S} \mathbin{//} P \triangleq (\langle \gamma \rangle \, (\langle \gamma' \rangle \, \mathtt{S})) \mathbin{//} P \qquad \langle A : \overline{\mathtt{D}} \rangle \, \mathtt{S} \mathbin{//} P \triangleq \langle A : \overline{\mathtt{D}}[A \mapsto P.A] \mathbin{//} P.A \rangle \, (\mathtt{S}[A \mapsto P.A] \mathbin{//} P)$$

For a single zipper $\langle A : \overline{\mathtt{D}} \rangle \, \mathtt{S}$, we do three things: (1) we strengthen inside the zipper context $A : \overline{\mathtt{D}}[A \mapsto P.A] \mathbin{//} P.A$, (2) we replace all *local accesses* by *zipper accesses* with a substitution, (3) we strengthen the zipped signature. Delayed strengthening on declarations is straightforward: only module declarations are strengthened. We define it on prefixes Q instead of paths P as delayed strengthening is also used inside zippers.

$$(\mathtt{val}\, x : \mathtt{u}) \mathbin{//} Q \triangleq \mathtt{val}\, x : \mathtt{u} \qquad (\mathtt{module}\, X : \mathtt{S}) \mathbin{//} Q \triangleq \mathtt{module}\, X : (\mathtt{S} \mathbin{//} Q.X)$$

$$(\mathtt{type}\, t = \mathtt{u}) \mathbin{//} Q \triangleq \mathtt{type}\, t = \mathtt{u} \qquad (\mathtt{module\ type}\ T = \mathtt{S}) \mathbin{//} Q \triangleq \mathtt{module\ type}\ T = \mathtt{S}$$

Environment extension Last, Figure 27 defines a helper binary operation $\uplus$ that we use to strengthen bindings as they enter the typing environment. We use it to maintain the invariant that all signatures entering the typing environment are strengthened. It is defined on functor parameters, declarations, and zipper contexts:

$$\Gamma \uplus (Y : \mathtt{S}) \triangleq \Gamma, Y : (\mathtt{S} \mathbin{//} Y) \qquad \Gamma \uplus (\mathtt{module}\, A.X : \mathtt{S}) \triangleq \Gamma, \mathtt{module}\, A.X : (\mathtt{S} \mathbin{//} A.X)$$

$$\Gamma \uplus (\langle A : \overline{\mathtt{D}} \rangle \, \gamma) \triangleq (\Gamma, A : (\overline{\mathtt{D}} \mathbin{//} A)) \uplus \gamma$$

5.3.4 Path typing, resolution and normalization

Since paths include projections and applications, which require recursive lookups and substitutions, their types are not immediate to deduce. Moreover, signatures in the typing environment may themselves be module type definitions that must be inlined to be analyzed. We use path resolution, path typing, and normalization for this purpose.

Path typing

The path typing judgment $\Gamma \vdash P : \mathbf{S}$ is defined in Figure 28 together with path resolution $\Gamma \vdash P \triangleright \mathtt{S}$,$\Gamma \vdash P \triangleright P'$ and discussed below. Intuitively, typing a path P returns *all* the environment information about the module at path P, including a potential zipper, an aliasing information with another path and the signature (not strengthened yet). To factor out a common pattern-match on the result of path typing, we also introduce *path resolution* $\Gamma \vdash P \triangleright \mathtt{S}$ to extract the module *content*, by dropping the zipper and aliasing information and forcing a shallow strengthening, and $\Gamma \vdash P \triangleright P'$ to extract the module *identity*, by dropping the zipper and signature.

Path resolution The two judgments are defined in Figure 28 by a single rule each. Rule Res-P-Id simply pattern-matches on the result of signature typing to return the aliasing information. Rule Res-P-Val pattern-matches on the result of signature typing both to extract $\tilde{\mathtt{S}}$ and to force it to be in head-normal (so that $\tilde{\mathtt{S}} \mathbin{/} P'$ is well-defined).

$$\textsc{Res-P-Val}\ \frac{\Gamma \vdash P : \mathtt{S} \qquad \mathtt{S} \triangleright \mathtt{S}'}{\Gamma \vdash P \triangleright \mathtt{S}'} \qquad \textsc{Res-P-Id}\ \frac{\Gamma \vdash P : \langle \boldsymbol{\gamma} \rangle \, (= P' < \tilde{\mathtt{S}})}{\Gamma \vdash P \triangleright P'}$$

$$\frac{Y : \mathbf{S} \in \Gamma}{\Gamma \vdash Y : \mathbf{S}}\ \textsc{Typ-P-Arg} \qquad \frac{\texttt{module}\ A.X : \mathbf{S} \in \Gamma}{\Gamma \vdash A.X : \mathbf{S}}\ \textsc{Typ-P-Module}$$

$$\frac{\Gamma \vdash P : \langle \boldsymbol{\gamma} \rangle\, \mathbf{S} \quad \boldsymbol{\gamma}(A) = \overline{\mathbf{D}}}{\Gamma \vdash P.A : \texttt{sig}\ \overline{\mathbf{D}}\ \texttt{end}}\ \textsc{Typ-P-Zip} \qquad \frac{\Gamma \vdash P : \mathbf{S} \quad \Gamma \vdash \mathbf{S} \downarrow \mathbf{S}'}{\Gamma \vdash P : \mathbf{S}'}\ \textsc{Typ-P-Norm}$$

$$\frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathbf{D}}\ \texttt{end} \quad \texttt{module}\ X : \mathbf{S} \in \overline{\mathbf{D}}}{\Gamma \vdash P.X : \mathbf{S}}\ \textsc{Typ-P-Proj}$$

$$\frac{\Gamma \vdash P \triangleright (Y : \mathtt{S}_a) \to \mathbf{S} \quad \Gamma \vdash P' : \mathbf{S}' \quad \Gamma \vdash \mathbf{S}' \leqslant \mathtt{S}_a}{\Gamma \vdash P(P') : \mathbf{S}[Y \mapsto P']}\ \textsc{Typ-P-AppA}$$

(a) Path typing – $\Gamma \vdash P : \mathtt{S}$

$$\frac{\Gamma \vdash P : \langle \boldsymbol{\gamma} \rangle\, (= P' < \tilde{\mathtt{S}})}{\Gamma \vdash P \triangleright P'}\ \textsc{Res-P-Id} \qquad \frac{\Gamma \vdash P : \langle \boldsymbol{\gamma} \rangle\, (= P' < \tilde{\mathtt{S}}) \quad \tilde{\mathtt{S}} \,/\, P' = \mathtt{S}'}{\Gamma \vdash P \triangleright \mathtt{S}'}\ \textsc{Res-P-Val}$$

(b) Path Resolution – $\Gamma \vdash P \triangleright P'$ and $\Gamma \vdash P \triangleright \tilde{\mathbf{S}}$

Figure 28: Path typing and path resolution

In both rules, we assume that zippers have been flattened and defaulting to the empty zipper if there is no zipper.

Path typing The two *leaf* rules that do actual lookups are Typ-P-Arg and Typ-P-Module:

$$\frac{Y : \mathbf{S} \in \Gamma}{\Gamma \vdash Y : \mathbf{S}}\ \textsc{Typ-P-Arg} \qquad \frac{\texttt{module}\ A.X : \mathbf{S} \in \Gamma}{\Gamma \vdash A.X : \mathbf{S}}\ \textsc{Typ-P-Module}$$

Normalization can be used at any point to inline module-type definitions (which is expected for reduction in types):

$$\frac{\Gamma \vdash P : \mathbf{S} \qquad \Gamma \vdash \mathbf{S} \downarrow \mathbf{S}'}{\Gamma \vdash P : \mathbf{S}'}\ \textsc{Typ-P-Norm}$$

Two rules use path resolution to analyze the signature in a premise: projection and application. In Rule Typ-P-Proj, we pattern-match on the resolution of the signature of P, and require that it is a structural signature. This may require any number of normalization steps. We omitted the self-reference of the signature of P since we know $\mathbf{S}$ has been strengthened and does not use its self-reference anymore. Typing a functor application (Rule Typ-P-AppA) requires the signature of P to be an applicative functor signature, which may again use any number of normalization steps followed by a resolution step. We also require the domain signature to be a supertype of the argument signature. We then return the codomain signature after substitution of the argument P for the parameter Y.

$$\frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathbf{D}}\ \texttt{end} \qquad \texttt{module}\ X : \mathbf{S} \in \overline{\mathbf{D}}}{\Gamma \vdash P.X : \mathbf{S}}\ \textsc{Typ-P-Proj}$$

$$\frac{\Gamma \vdash P \triangleright (Y : \mathtt{S}_a) \to \mathbf{S} \qquad \Gamma \vdash P' : \mathbf{S}' \qquad \Gamma \vdash \mathbf{S}' \leqslant \mathtt{S}_a}{\Gamma \vdash P(P') : \mathbf{S}[Y \mapsto P']}\ \textsc{Typ-P-AppA}$$

Finally, rule Typ-P-Zip accesses a zipper context through its self-reference. The signature $\texttt{sig}_A\ \overline{\mathbf{D}}\ \texttt{end}$ of $P.A$ need not be put back inside the zipper context $\boldsymbol{\gamma}$, since the signature $\langle \boldsymbol{\gamma} \rangle\, \mathbf{S}$, and hence the declarations $\overline{\mathbf{D}}$, have been strengthened and no longer depend on the zipper context $\boldsymbol{\gamma}$, but only on the typing environment Γ.

Notice that path typing is not deterministic: there may be two signatures $\mathbf{S}_1$ and $\mathbf{S}_2$ such that $\Gamma \vdash P : \mathbf{S}_1$ and $\Gamma \vdash P : \mathbf{S}_2$, where $\mathbf{S}_1$ and $\mathbf{S}_2$ are not α-equivalent. This comes from the floating normalization rule, that may be used any number of times at any depth.

$$\textsc{Norm-S-Zip}\ \frac{\Gamma \uplus \gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash \langle\gamma\rangle\,\mathtt{S} \downarrow \langle\gamma\rangle\,\mathtt{S}'} \qquad \textsc{Norm-S-Trans-Some}\ \frac{\Gamma \vdash \mathtt{S} \downarrow (= P' < \mathtt{S}')}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')} \qquad \textsc{Norm-S-Trans-None}\ \frac{\Gamma \vdash P \triangleright P' \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')}$$

$$\textsc{Norm-S-LocalModType}\ \frac{\texttt{module type}\ A.T = \mathtt{S} \in \Gamma}{\Gamma \vdash A.T \downarrow \mathtt{S}} \qquad \textsc{Norm-S-PathModType}\ \frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathtt{D}}\ \texttt{end} \qquad \texttt{module type}\ T = \mathtt{S} \in \overline{\mathtt{D}}}{\Gamma \vdash P.T \downarrow \mathtt{S}}$$

$$\textsc{Norm-Typ-Res}\ \frac{\Gamma \vdash P \triangleright P'}{\Gamma \vdash P.t \downarrow P'.t} \qquad \textsc{Norm-Typ-Local}\ \frac{\texttt{type}\ A.t = \mathtt{u} \in \Gamma}{\Gamma \vdash A.t \downarrow \mathtt{u}} \qquad \textsc{Norm-Typ-Path}\ \frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathtt{D}}\ \texttt{end} \qquad \texttt{type}\ t = \mathtt{u} \in \overline{\mathtt{D}}}{\Gamma \vdash P.t \downarrow \mathtt{u}}$$

Figure 29: Signature and type normalization – $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$

Normalization

The judgment $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$ *allows* the (head) normalization of a signature $\mathtt{S}$ into $\mathtt{S}'$, which inlines module-type definitions, but only one step at a time. Hence, to achieve the head normal form, we may call normalization repeatedly. It is defined in Figure 29 and discussed below.

Rule Norm-S-Zip normalizes the zipped signature. We never normalize the zipper context itself, as we will first access the zipper context and normalize the result afterwards.

$$\textsc{Norm-S-Zip}\ \frac{\Gamma \uplus \gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash \langle\gamma\rangle\,\mathtt{S} \downarrow \langle\gamma\rangle\,\mathtt{S}'}$$

Rules Norm-S-Trans-Some and Norm-S-Trans-None allows normalization under a transparent ascription. If the (partial) normal form of $\mathtt{S}$ is itself a transparent ascription, we return it as is; otherwise, we return its strengthened version by the resolved path P'.

$$\textsc{Norm-S-Trans-Some}\ \frac{\Gamma \vdash \mathtt{S} \downarrow (= P' < \mathtt{S}')}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')} \qquad \textsc{Norm-S-Trans-None}\ \frac{\Gamma \vdash P \triangleright P' \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')}$$

Finally, rules Norm-S-LocalModType and Norm-S-PathModType expand a module-type definition:

$$\textsc{Norm-S-LocalModType}\ \frac{\texttt{module type}\ A.T = \mathtt{S} \in \Gamma}{\Gamma \vdash A.T \downarrow \mathtt{S}} \qquad \textsc{Norm-S-PathModType}\ \frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathtt{D}}\ \texttt{end} \qquad \texttt{module type}\ T = \mathtt{S} \in \overline{\mathtt{D}}}{\Gamma \vdash P.T \downarrow \mathtt{S}}$$

The judgment $\Gamma \vdash P.t \downarrow \mathtt{u}$ allows normalization of types. Rules Norm-Typ-Res allows the resolution of the path P while rules Norm-Typ-Local and Norm-Typ-Path inlined the type definition $Q.t$.

5.3.5 Subtyping

The subtyping judgment $\Gamma \vdash \mathtt{S} \leq \tilde{\mathtt{S}}$ is only defined when the both sides are wellformed, and, until Section 5.4, when the right-hand side does not have zippers. The rules are given in Figure 30 and discussed below. Overall, subtyping is just a combination of structural subtyping of records, subtyping of functions (contra-variant on the domain, covariant on the codomain) and instantiation of abstract types. We actually define a judgment that is parameterized by a flag to be *code-free* or *general*:

$$\textsc{Sub-S-Norm}\ \frac{\Gamma \vdash \mathtt{S}_1 \downarrow \mathtt{S}_1' \qquad \Gamma \vdash \mathtt{S}_2 \downarrow \mathtt{S}_2' \qquad \Gamma \vdash \mathtt{S}_1' \leq \mathtt{S}_2'}{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2} \qquad \textsc{Sub-S-Zipper}\ \frac{\Gamma \uplus \gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash \langle\gamma\rangle\, \mathtt{S}_1 \leq \mathtt{S}_2} \qquad \textsc{Sub-S-TrAscr}\ \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathtt{S}_1) \leq (= P < \mathtt{S}_2)}$$

$$\textsc{Sub-S-LooseAlias}\ \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2}{\Gamma \vdash (= P < \mathtt{S}_1) \leq \mathtt{S}_2} \qquad \textsc{Sub-S-FctG}\ \frac{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash () \to \mathtt{S}_1 \leq () \to \mathtt{S}_2} \qquad \textsc{Sub-S-FctA}\ \frac{\Gamma \vdash \mathtt{S}_{a_2} \leq \mathtt{S}_{a_1} \qquad \Gamma \uplus Y : \mathtt{S}_{a_2} \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash (Y : \mathtt{S}_{a_1}) \to \mathtt{S}_1 \leq (Y : \mathtt{S}_{a_2}) \to \mathtt{S}_2}$$

$$\textsc{Sub-S-Sig}\ \frac{\overline{\mathtt{D}_0} \sqsubset_{\leq} \overline{\mathtt{D}_1} \qquad \Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \overline{\mathtt{D}_0} \,/\!/\, A \leq \overline{\mathtt{D}_2}}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}_1}\ \mathtt{end} \leq \mathtt{sig}_A\ \overline{\mathtt{D}_2}\ \mathtt{end}} \qquad \textsc{Sub-S-DynEq}\ \frac{\Gamma \vdash \mathtt{S} \lesssim \mathtt{S}' \qquad \Gamma \vdash \mathtt{S}' \lesssim \mathtt{S}}{\Gamma \vdash \mathtt{S} \approx \mathtt{S}'} \qquad \textsc{Sub-T-Norm}\ \frac{\Gamma \vdash \mathtt{u}_1 \downarrow \mathtt{u} \qquad \Gamma \vdash \mathtt{u}_2 \downarrow \mathtt{u}}{\Gamma \vdash \mathtt{u}_1 \leq \mathtt{u}_2}$$

$$\textsc{Sub-D-Mod}\ \frac{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash \mathtt{module}\, X : \mathtt{S}_1 \leq \mathtt{module}\, X : \mathtt{S}_2} \qquad \textsc{Sub-D-Val}\ \frac{\Gamma \vdash \mathtt{u}_1 \leq \mathtt{u}_2}{\Gamma \vdash \mathtt{val}\, x : \mathtt{u}_1 \leq \mathtt{val}\, x : \mathtt{u}_2}$$

$$\textsc{Sub-D-Modtype}\ \frac{\Gamma \vdash \mathtt{S}_1 \approx \mathtt{S}_2}{\Gamma \vdash \mathtt{module\ type}\ T = \mathtt{S}_1 \leq \mathtt{module\ type}\ T = \mathtt{S}_2} \qquad \textsc{Sub-D-Type}\ \frac{\Gamma \vdash \mathtt{u}_1 \approx \mathtt{u}_2}{\Gamma \vdash \mathtt{type}\, t = \mathtt{u}_1 \leq \mathtt{type}\, t = \mathtt{u}_2}$$

Figure 30: Subtyping – $\Gamma \vdash \mathtt{S} \leq \tilde{\mathtt{S}}$

- $\Gamma \vdash \mathtt{S} \lesssim \tilde{\mathtt{S}}$ is *code-free* subtyping. We define code-free equivalence as its kernel.
- $\Gamma \vdash \mathtt{S} \leqslant \tilde{\mathtt{S}}$ is *general* subtyping, possibly not code-free

The two differ only in the *width* rule for structural signatures.

$$\textsc{Sub-S-Sig}\ \frac{\overline{\mathtt{D}_0} \sqsubset_{\leq} \overline{\mathtt{D}_1} \qquad \Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \overline{\mathtt{D}_0} \,/\!/\, A \leq \overline{\mathtt{D}_2}}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}_1}\ \mathtt{end} \leq \mathtt{sig}_A\ \overline{\mathtt{D}_2}\ \mathtt{end}}$$

In both cases, a certain subset $\overline{\mathtt{D}_0}$ of the left-hand side fields $\overline{\mathtt{D}_1}$ is chosen to be compared against the right-hand side fields $\overline{\mathtt{D}_2}$. This comparison is done *field-by-field*, in parallel, not as a telescope. As we push $\overline{\mathtt{D}_1}$ in the context, they get strengthened. The subset set of declarations $\overline{\mathtt{D}_0}$ is also strengthened to correctly refer to $\overline{\mathtt{D}_1}$, possibly deeply.

The main subtyping relation is $\leqslant$ defined as $\leqslant_{\subseteq}$, i.e., using $\subseteq$ for $\sqsubset_{\leqslant}$. Namely, $\overline{\mathtt{D}_0}$ can be any subset of $\overline{\mathtt{D}_1}$ where fields may appear in a different order. This relation is (usually) not code-free, as ML modules are compiled with a *static dispatch* scheme for performance (see Section 2.2.3). The code-free version of subtyping, $\lesssim$, is defined by setting the relation $\sqsubset_{\lesssim}$ as:

$$\overline{\mathtt{D}_0} \sqsubset_{\lesssim} \overline{\mathtt{D}_1} \triangleq \overline{\mathtt{D}_0} \subseteq \overline{\mathtt{D}_1} \wedge \mathsf{dyn}(\overline{\mathtt{D}_0}) = \mathsf{dyn}(\overline{\mathtt{D}_1})$$

where $\mathsf{dyn}(\overline{\mathtt{D}})$ returns the subsequence of $\overline{\mathtt{D}}$ composed of dynamic fields (modules and values) only (appearing in the same order). We define the dynamic part $\lfloor \mathtt{S} \rfloor$ of a signature $\mathtt{S}$ as a pseudo (untyped) signature that erases all parts of signatures describing static fields:

$$\begin{array}{rcl} \lfloor (= P < \mathtt{S}) \rfloor & \triangleq & \lfloor \mathtt{S} \rfloor \\ \lfloor () \to \mathtt{S} \rfloor & \triangleq & () \to \lfloor \mathtt{S} \rfloor \\ \lfloor (Y : \mathtt{S}_a) \to \mathtt{S} \rfloor & \triangleq & \lfloor \mathtt{S}_a \rfloor \to \lfloor \mathtt{S} \rfloor \\ \lfloor \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} \rfloor & \triangleq & \mathtt{sig}\ \lfloor \overline{\mathtt{D}} \rfloor\ \mathtt{end} \\ \lfloor \langle\gamma\rangle\, \mathtt{S} \rfloor & \triangleq & \lfloor \mathtt{S} \rfloor \end{array} \qquad \begin{array}{rcl} \lfloor \mathtt{val}\, x : \_ \rfloor & \triangleq & \mathtt{val}\, x \\ \lfloor \mathtt{module}\, X : \mathtt{S} \rfloor & \triangleq & \mathtt{module}\, X : \lfloor \mathtt{S} \rfloor \\ \lfloor \mathtt{type}\, t = \_ \rfloor & \triangleq & \varnothing \\ \lfloor \mathtt{module\ type}\ T = \_ \rfloor & \triangleq & \varnothing \end{array}$$

Dynamic parts of signatures are then compared syntactically.

The other rules are *flag-polymorphic*, i.e., they are the same for both modes of subtyping and use the same flag in the premises and conclusion. They must be read by case analysis on the left-hand side signature. First, normalization is injected anywhere by Rule Sub-S-Norm:

$$\textsc{Sub-S-Norm}\quad \frac{\Gamma \vdash \mathtt{S}_1 \downarrow \mathtt{S}'_1 \qquad \Gamma \vdash \mathtt{S}_2 \downarrow \mathtt{S}'_2 \qquad \Gamma \vdash \mathtt{S}'_1 \leq \mathtt{S}'_2}{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}$$

For example, there is no rule matching a signature $Q.T$ on the left-hand side, as normalization allows inlining the definition before checking for subtyping. A zipper may only occur on the left-hand side. Rule Sub-S-Zipper pushes the zipper context in the typing environment (as the signature $\mathtt{S}_1$ may not be transparent) and pursues with subtyping.

$$\textsc{Sub-S-Zipper}\quad \frac{\Gamma \uplus \gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash \langle\gamma\rangle\, \mathtt{S}_1 \leq \mathtt{S}_2}$$

For other cases, we require the left-hand side to be in head normal form. When the left-hand side is a transparent ascription, the right-hand side may also be a transparent ascription, in which case, we check subtyping between the respective signatures, but after pushing strengthening one level-down, lazily (Sub-S-TrAscr). Otherwise, we drop the transparent ascription from the left-hand side, which amounts to a loss of transparency, hence increase abstraction, as allowed by subtyping (Sub-S-LooseAlias).

$$\textsc{Sub-S-TrAscr}\quad \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathtt{S}_1) \leq (= P < \mathtt{S}_2)} \qquad \textsc{Sub-S-LooseAlias}\quad \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2}{\Gamma \vdash (= P < \mathtt{S}_1) \leq \mathtt{S}_2}$$

In the remaining cases, the left-hand side is a head value form and the right-hand side must have the same shape. Functor types are contravariant:

$$\textsc{Sub-S-FctG}\quad \frac{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash () \to \mathtt{S}_1 \leq () \to \mathtt{S}_2} \qquad \textsc{Sub-S-FctG}\quad \frac{\Gamma \vdash \mathtt{S}_{a_2} \leq \mathtt{S}_{a_1} \qquad \Gamma \uplus Y : \mathtt{S}_{a_2} \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash (Y : \mathtt{S}_{a_1}) \to \mathtt{S}_1 \leq (Y : \mathtt{S}_{a_2}) \to \mathtt{S}_2}$$

Rule Sub-S-DynEq defines code-free equivalence, which is a separate judgment. Subtyping itself is not code-free as it may remove or reorder dynamic fields at runtime.

Declarations

Normalization Subtyping uses two other helper judgments, for type and declaration subtyping. There is a single rule Sub-T-Norm for subtyping of core-language types that injects head type normalization into the subtyping relation, which is the pending of Rule Sub-S-Norm for core-language types. In fact, this rule should also be made available in the subtyping relation of the core language, which should be a congruent pre-order.

$$\textsc{Sub-T-Norm}\quad \frac{\Gamma \vdash \mathtt{u}_1 \downarrow \mathtt{u} \qquad \Gamma \vdash \mathtt{u}_2 \downarrow \mathtt{u}}{\Gamma \vdash \mathtt{u}_1 \leq \mathtt{u}_2}$$

Values and modules For module declarations (Rule Sub-D-Mod), we just require subtyping covariantly. Rule Sub-D-Val for core language values is similar, requiring subtyping in the core language. In OCaml, this would reduce to core-language type-scheme specialization, which we haven't formalized:

$$\textsc{Sub-D-Mod}\quad \frac{\Gamma \vdash \mathtt{S}_1 \leq \mathtt{S}_2}{\Gamma \vdash \mathtt{module}\, X : \mathtt{S}_1 \leq \mathtt{module}\, X : \mathtt{S}_2} \qquad \textsc{Sub-D-Val}\quad \frac{\Gamma \vdash \mathtt{u}_1 \leq \mathtt{u}_2}{\Gamma \vdash \mathtt{val}\, x : \mathtt{u}_1 \leq \mathtt{val}\, x : \mathtt{u}_2}$$

Types and module-types Since module-types may be used in both covariant and contravariant positions, the rule Sub-D-Modtype requests subtyping in both directions. Moreover, since this subtyping could occur deeply inside terms, we use code-free type equivalence, defined as the kernel of code-free subtyping:

$$\textsc{Sub-S-DynEq}\quad \frac{\Gamma \vdash \mathtt{S} \lesssim \mathtt{S}' \qquad \Gamma \vdash \mathtt{S}' \lesssim \mathtt{S}}{\Gamma \vdash \mathtt{S} \approx \mathtt{S}'}$$

Notice that if signatures were fully inlined, subtyping would never see the names of definitions but their original inlined expansion and covariance would suffice.

$$\textsc{Sub-D-Modtype}\quad \frac{\Gamma \vdash \mathtt{S}_1 \approx \mathtt{S}_2}{\Gamma \vdash \mathtt{module\ type}\ T = \mathtt{S}_1 \leq \mathtt{module\ type}\ T = \mathtt{S}_2} \qquad \textsc{Sub-D-Type}\quad \frac{\Gamma \vdash \mathtt{u}_1 \approx \mathtt{u}_2}{\Gamma \vdash \mathtt{type}\, t = \mathtt{u}_1 \leq \mathtt{type}\, t = \mathtt{u}_2}$$

The same remark applies to core-language type fields, which are also used as definitions and may appear both co- and contra-variantly. Hence, Rule Sub-D-Type requires $\Gamma \vdash \mathtt{u}_1 \approx \mathtt{u}_2$ which means code-free subtyping in both directions, i.e., $\Gamma \vdash \mathtt{u}_1 \leq \mathtt{u}_2$ and $\Gamma \vdash \mathtt{u}_2 \leq \mathtt{u}_1$.

No rule for abstract types ? The reader might wonder where the rule that allows subtyping of concrete type with an abstract type is, of the form $\Gamma \vdash_A \mathtt{type}\, t = \mathtt{int} \leq \mathtt{type}\, t$. Actually, our encoding of abstract type as equal to themselves removes the need for such rule. Let us look at an example subtyping derivation between a signature with concrete type and a signature with an abstract type:

$$\cfrac{\cfrac{(A : \mathtt{type}\, t = \mathtt{int}) \vdash \mathtt{int} \approx A.t}{(A : \mathtt{type}\, t = \mathtt{int}) \vdash_A \mathtt{type}\, t = \mathtt{int} \leq \mathtt{type}\, t = A.t}\ \textsc{Sub-D-Type}}{\varnothing \vdash \mathtt{sig}_A\ \mathtt{type}\ t = \mathtt{int}\ \mathtt{end} \leq \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end}}\ \textsc{Sub-S-Sig}$$

The key is that the structural signatures rule Sub-S-Sig puts the declarations of the left-hand side signature in the context before subtyping the fields. Therefore, the typing environment contains the declaration $A : \mathtt{type}\, t = \mathtt{int}$. By normalization, it makes $A.t$ and $\mathtt{int}$ equivalent.

Subtyping and strengthening

Strengthening adds new equalities, both at the type and module levels. Therefore, it should not limit subtyping when used on the left-hand side. That is, we have[8] $\Gamma \vdash \mathtt{S} \,//\, P \leq \mathtt{S}$, which relies on the fact that $\Gamma \vdash \mathtt{S} \,/\, P \leq \mathtt{S}$. Is proved by a straightforward induction, where the core argument is illustrated by the derivation above.

Optimization

Judgments $\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2$ and $\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2 \,/\, P$ (when both sides are wellformed) are in fact equivalent. Intuitively, a derivation of the former may abstract some types appearing in $\mathtt{S}_1 \,/\, P$, but never has to, i.e., the same derivation could be reproduced without any abstraction. Therefore, Rule Sub-S-LooseAlias could be replaced by rule Sub-S-LooseAlias-Opt:

$$\textsc{Sub-S-LooseAlias-Opt}\quad \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leq \mathtt{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathtt{S}_1) \leq \mathtt{S}_2} \qquad \textsc{Sub-S-Sig-Opt}\quad \frac{\overline{\mathtt{D}_0} \sqsubset_{\leq} \overline{\mathtt{D}_1} \qquad \Gamma \vdash \overline{\mathtt{D}_0} \leq \overline{\mathtt{D}_2}}{\Gamma \vdash \mathtt{sig}\ \overline{\mathtt{D}_1}\ \mathtt{end} \leq \mathtt{sig}\ \overline{\mathtt{D}_2}\ \mathtt{end}}$$

[8] Assuming that both sides are wellformed, i.e., $\Gamma \vdash \mathtt{S} : \mathtt{wf}$ and $\Gamma \vdash \mathtt{S} \,//\, P : \mathtt{wf}$. The definition of wellformedness is given in Section 5.3.6

$$\textsc{Typ-S-ModType}\ \frac{\Gamma \vdash \tilde{Q}.T : \mathtt{S}}{\Gamma \vdash \tilde{Q}.T} \qquad \textsc{Typ-S-GenFct}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash () \to \mathtt{S}} \qquad \textsc{Typ-S-AppFct}\ \frac{\Gamma \vdash \mathtt{S}_a \qquad \Gamma \uplus (Y : \mathtt{S}_a) \vdash \mathtt{S}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S}}$$

$$\textsc{Typ-S-Ascr}\ \frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \tilde{P} : \mathbf{S} \qquad \Gamma \vdash \mathbf{S} \leqslant \mathtt{S}}{\Gamma \vdash (= \tilde{P} < \mathtt{S})} \qquad \textsc{Typ-S-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}}$$

(a) Signatures rules

$$\textsc{Typ-D-Val}\ \frac{\Gamma \vdash \tilde{\mathtt{u}}}{\Gamma \vdash_A (\mathtt{val}\, x : \tilde{\mathtt{u}})} \qquad \textsc{Typ-D-Type}\ \frac{\Gamma \vdash \tilde{\mathtt{u}}}{\Gamma \vdash_A (\mathtt{type}\, t = \tilde{\mathtt{u}})} \qquad \textsc{Typ-D-TypeAbs}\ \Gamma \vdash_A (\mathtt{type}\, t = A.t) \qquad \textsc{Typ-D-Mod}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S})}$$

$$\textsc{Typ-D-ModType}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S})} \qquad \textsc{Typ-D-Empty}\ \Gamma \vdash_A \varnothing \qquad \textsc{Typ-D-Seq}\ \frac{\Gamma \vdash_A \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}_0 \vdash_A \overline{\mathtt{D}}}{\Gamma \vdash_A (\mathtt{D}_0, \overline{\mathtt{D}})}$$

(b) Declarations rules

Figure 31: Signature and declaration typing (all signatures are $\tilde{\mathtt{S}}$) – $\Gamma \vdash \tilde{\mathtt{S}}$

We may then add Rule Sub-S-Optim, which is an instance of Sub-S-Sig that can be used when neither side uses its self-reference: we can avoid pushing useless information in the typing environment. Overall, this comes from the fact that we use subtyping both on signatures coming from the typing environment (fully strengthened) and signatures coming from inference (not strengthened) yet. We need to enrich the typing environment only for the latter.

5.3.6 Signature typing

Source signatures provided by users are not necessarily well formed, and thus must be typed, using the judgment $\Gamma \vdash \tilde{\mathtt{S}}$, defined in Figure 31. The resulting source signature $\tilde{\mathtt{S}}$ is zipper free. Since this is also enforced by not having a typing rule for zippers, we have not enforced the syntactic subcategories and just use $\mathtt{S}$ and $\mathtt{D}$ for signatures and declarations, for the sake of readability. The signature $\tilde{\mathtt{S}}$ should not have zipper-context accesses either, which is enforced by using the restricted syntactic categories $\tilde{Q}$ and $\tilde{P}$ for paths.

Most of the typing rules are straightforward wellformedness checks. The only involved rule is Typ-S-Ascr. It checks that the signature $\mathbf{S}$ of path P is a subtype of the signature $\tilde{\mathtt{S}}$.

$$\textsc{Typ-S-Ascr}\ \frac{\Gamma \vdash \tilde{\mathtt{S}} \qquad \Gamma \vdash \tilde{P} : \mathbf{S} \qquad \Gamma \vdash \mathbf{S} \leqslant \mathtt{S}}{\Gamma \vdash (= \tilde{P} < \mathtt{S})}$$

If we extend the signature language with **`include`**, **`open`** or the constructs discussed in Section 6.2.2, signature typing would become a light elaboration, of the form $\Gamma \vdash \mathtt{S} : \mathtt{S}'$.

Wellformedness

If we have not defined a signature wellformedness judgment earlier, it is because we could not: checking a signature requires to typecheck paths and subtyping. Adding wellformedness preconditions everywhere would have made the system more recursively intertwined. Instead, we define it now as an extension of signature typing, and show that it is maintained throughout

every judgment. We define $\Gamma \vdash \mathtt{S} : \mathsf{wf}$ with the same rules as signature typing but (1) removing the condition that paths do not contain zippers, and (2) adding a rule for zippers:

$$\textsc{WF-S-Zip}\quad \frac{\Gamma \vdash_A \overline{\mathtt{D}} : \mathsf{wf} \qquad \Gamma \uplus \overline{\mathtt{D}} \vdash \langle\gamma\rangle\, \mathtt{S} : \mathsf{wf}}{\Gamma \vdash \langle A : \overline{\mathtt{D}}\,;\gamma\rangle\, \mathtt{S} : \mathsf{wf}}$$

We extend wellformedness to environment wellformedness, written $\vdash \Gamma : \mathsf{wf}$, as a simple fold of wellformedness of all components. From there, an immediate induction shows that, assuming $\vdash \Gamma : \mathsf{wf}$, we have:

$$\Gamma \vdash \mathtt{S} : \mathsf{wf} \land \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}' \implies \Gamma \vdash \mathtt{S}' : \mathsf{wf} \qquad \Gamma \vdash P : \mathtt{S} \implies \Gamma \vdash \mathtt{S} : \mathsf{wf} \qquad \Gamma \vdash \mathtt{S} \implies \Gamma \vdash \mathtt{S} : \mathsf{wf}$$

Subtyping is only defined on wellformed signatures.

Alternative definition Similarly, we could have defined both wellformedness and signature typing as a single judgment parameterized by a flag to indicate if it enforces no zipper accesses or not.

5.3.7 Module typing

The rules for the module typing judgment $\Gamma \vdash^{\diamondsuit} \mathtt{M} : \mathtt{S}$ are given in Figure 32 and discussed below.

Modes The $\diamondsuit$ symbol is a metavariable for modes that ranges over the applicative (or *transparent*) mode $\triangledown$ and the generative (or *opaque*) mode $\blacktriangledown$. Rule Typ-M-Mode allows to always consider an applicative expression as a generative one: This is a floating rule that can be applied at any time. Judgments for pure module expressions can be treated either as applicative or generative, hence they use the $\diamondsuit$ metavariable. Many rules use the same metavariable $\diamondsuit$ in premises and conclusion, which then stand for the same mode. This implies that if the premise can only be proved in generative mode, it will also be the case for the conclusion.

$$\textsc{Typ-M-Mode}\quad \frac{\Gamma \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}$$

Normalization Normalization can be applied at any point. This is required by other rules that pattern-match on the signature of sub-expressions.

$$\textsc{Typ-M-Norm}\quad \frac{\Gamma \vdash^{\diamondsuit} \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash^{\diamondsuit} \mathtt{M} : \mathtt{S}'}$$

Paths Typing a path $\tilde{P}$ (which should not contain zipper-context accesses) as a module expression (Rule Typ-M-Path) calls the path typing rule defined earlier. It always gives a transparent signature $(= P' < \mathtt{S})$, possibly with a zipper. However, we return the "more recent" identity $(= \tilde{P} < \mathtt{S})$, as this is probably the one the user would like to see; besides, the older identities can always be recovered by normalization, while the reverse is not possible. We drop the zipper as, thanks to strengthening, it is not accessed by the result signature.

$$\textsc{Typ-M-Path}\quad \frac{\Gamma \vdash \tilde{P} : \langle\gamma\rangle\, (= P' < \mathtt{S})}{\Gamma \vdash^{\diamondsuit} \tilde{P} : (= \tilde{P} < \mathtt{S})}$$

$$\textsc{Typ-M-Norm}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}'} \qquad \textsc{Typ-M-Mode}\ \frac{\Gamma \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}} \qquad \textsc{Typ-M-Ascr}\ \frac{\Gamma \vdash \tilde{\mathtt{S}} \qquad \Gamma \vdash \tilde{P} : \mathbf{S} \qquad \Gamma \vdash \mathbf{S} \leqslant \tilde{\mathtt{S}}}{\Gamma \vdash^{\diamond} (\tilde{P} : \tilde{\mathtt{S}}) : \tilde{\mathtt{S}}}$$

$$\textsc{Typ-M-Path}\ \frac{\Gamma \vdash \tilde{P} : \langle\gamma\rangle\,(= P' < \mathtt{S})}{\Gamma \vdash^{\diamond} \tilde{P} : (= \tilde{P} < \mathtt{S})} \qquad \textsc{Typ-M-FctG}\ \frac{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\diamond} () \to \mathtt{M} : () \to \mathtt{S}} \qquad \textsc{Typ-M-AppG}\ \frac{\Gamma \vdash P \triangleright () \to \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} P() : \mathtt{S}}$$

$$\textsc{Typ-M-FctA}\ \frac{\Gamma \vdash \mathtt{S}_a \qquad \Gamma \uplus Y : \mathtt{S}_a \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\diamond} (Y : \mathtt{S}_a) \to \mathtt{M} : (Y : \mathtt{S}'_a) \to \mathtt{S}} \qquad \textsc{Typ-M-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}}$$

$$\textsc{Typ-M-ProjT}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \langle\gamma\rangle\,(= P < \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end})}{\Gamma \vdash^{\diamond} \mathtt{M}.X : \langle\gamma\rangle\,(= P.X < \mathtt{S}[A \mapsto P])} \qquad \textsc{Typ-M-ProjA}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \langle\gamma\rangle\ \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash^{\diamond} \mathtt{M}.X : \langle\gamma\,;A : \overline{\mathtt{D}}\rangle\,\mathtt{S}}$$

(a) $\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}$ – Typing rules for module expressions

$$\textsc{Typ-B-Seq}\ \frac{\Gamma \vdash_A^{\diamond} \mathtt{B}_0 : \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}_0 \vdash_A^{\diamond} \overline{\mathtt{B}} : \overline{\mathtt{D}}}{\Gamma \vdash_A^{\diamond} \mathtt{B}_0, \overline{\mathtt{B}} : \mathtt{D}_0, \overline{\mathtt{D}}} \qquad \textsc{Typ-B-Type-Bind}\ \frac{\Gamma \vdash \tilde{\mathtt{u}}}{\Gamma \vdash_A^{\diamond} (\mathtt{type}\ t = \tilde{\mathtt{u}}) : (\mathtt{type}\ t = \tilde{\mathtt{u}})} \qquad \textsc{Typ-B-Empty}\ \Gamma \vdash_A^{\diamond} \varnothing : \varnothing$$

$$\textsc{Typ-B-AbsType}\ \Gamma \vdash_A^{\diamond} (\mathtt{type}\ t = A.t) : (\mathtt{type}\ t = A.t) \qquad \textsc{Typ-B-Let}\ \frac{\Gamma \vdash^{\diamond} \mathtt{e} : \mathtt{u}}{\Gamma \vdash_A^{\diamond} (\mathtt{let}\ x = \mathtt{e}) : (\mathtt{val}\ x : \mathtt{u})}$$

$$\textsc{Typ-B-Mod}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}}{\Gamma \vdash_A^{\diamond} (\mathtt{module}\ X = \mathtt{M}) : (\mathtt{module}\ X : \mathtt{S})} \qquad \textsc{Typ-B-ModType}\ \frac{\Gamma \vdash \tilde{\mathtt{S}}}{\Gamma \vdash_A^{\diamond} (\mathtt{module\ type}\ T = \tilde{\mathtt{S}}) : (\mathtt{module\ type}\ T = \tilde{\mathtt{S}})}$$

(b) $\Gamma \vdash_A^{\diamond} \mathtt{B} : \mathtt{D}$ – Typing rules for bindings

Figure 32: Typing rules for module expressions and bindings

Ascription A signature ascription $(\tilde{P} : \tilde{\mathtt{S}})$ has the signature $\tilde{\mathtt{S}}$ provided it is indeed a supertype of the signature $\mathtt{S}$ of $\tilde{P}$ alone, as stated by the Rule Typ-M-Ascr below. This ascription is opaque since it returns $\tilde{\mathtt{S}}$ un-strengthened by $\tilde{P}$.

$$\textsc{Typ-M-Ascr}\ \frac{\Gamma \vdash \tilde{\mathtt{S}} \qquad \Gamma \vdash \tilde{P} : \mathbf{S} \qquad \Gamma \vdash \mathbf{S} \leqslant \tilde{\mathtt{S}}}{\Gamma \vdash^{\diamond} (\tilde{P} : \tilde{\mathtt{S}}) : \tilde{\mathtt{S}}}$$

Functors A generative functor Typ-M-FctG is just an evaluation barrier: the functor itself is applicative while the body is generative. Correspondingly, applying a generative functor (Rule Typ-M-AppG), which amounts to evaluating its body, is then generative:

$$\textsc{Typ-M-FctG}\ \frac{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\diamond} () \to \mathtt{M} : () \to \mathtt{S}} \qquad \textsc{Typ-M-AppG}\ \frac{\Gamma \vdash P \triangleright () \to \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} P() : \mathtt{S}}$$

Here, we implicitly rely on the renaming of self-references to make sure that the newly generated module $P()$ has a fresh self-reference if $\mathtt{S}$ is a structural signature. An applicative functor

definition requires the body to be in applicative mode:

$$\textsc{Typ-M-FctA}\ \frac{\Gamma \vdash \mathtt{S}_a \qquad \Gamma \uplus Y : \mathtt{S}_a \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\diamondsuit} (Y : \mathtt{S}_a) \to \mathtt{M} : (Y : \mathtt{S}'_a) \to \mathtt{S}}$$

There is no application rule for applicative functors, as it is already covered by paths.

Structures and bindings Rule Typ-M-Str for structures delays the work to the typing rules for bindings, which carry the self-variable A of the structure as an annotation that should be chosen fresh for the context Γ. The remaining rules are for typing of bindings, which work as expected. In particular, Rule Typ-B-Seq pushes the declaration $A.\mathtt{D}_0$ into the context while typing the declarations $\overline{\mathtt{D}}$, much as Typ-D-Seq for signatures:

$$\textsc{Typ-M-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}} \qquad \textsc{Typ-B-Seq}\ \frac{\Gamma \vdash_A^{\diamondsuit} \mathtt{B}_0 : \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}_0 \vdash_A^{\diamondsuit} \overline{\mathtt{B}} : \overline{\mathtt{D}}}{\Gamma \vdash_A^{\diamondsuit} \mathtt{B}_0, \overline{\mathtt{B}} : \mathtt{D}_0, \overline{\mathtt{D}}}$$

Projection The first rule of projection is Typ-M-ProjT, when the module has a transparent signature:

$$\textsc{Typ-M-ProjT}\ \frac{\Gamma \vdash^{\diamondsuit} \mathtt{M} : \langle\gamma\rangle\ (= P < \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end})}{\Gamma \vdash^{\diamondsuit} \mathtt{M}.X : \langle\gamma\rangle\ (= P.X < \mathtt{S}[A \mapsto P])}$$

In such case, there is no need to introduce a new zipper, substituting A by $P.A$ is enough. Finally, Rule Typ-M-ProjA for typing a projection $\mathtt{M}.X$ where the signature is not transparent, is the key rule that leverages zippers:

$$\textsc{Typ-M-ProjA}\ \frac{\Gamma \vdash^{\diamondsuit} \mathtt{M} : \langle\gamma\rangle\ \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash^{\diamondsuit} \mathtt{M}.X : \langle \gamma\,;\, A : \overline{\mathtt{D}} \rangle\ \mathtt{S}}$$

Intuitively, it just returns the signature $\mathtt{S}'$ of the field X of the signature $\mathtt{S}$ of $\mathtt{M}$ zipped around the initial fields $\overline{\mathtt{D}}$ of $\mathtt{S}$ appearing before the field X. Still, we have to consider that the signature of $\mathtt{M}$ may itself be in a zipper context γ, which is then composed with the zipper context formed of the initial fields $\overline{\mathtt{D}}$, resulting in $\gamma\,;\, A : \overline{\mathtt{D}}$. This fixes the self-reference A, which is no longer α-convertible.

Missing rules There is no floating subtyping rule. It would not make sense for general subtyping, which is not code free, but we also did not include a rule for code-free subtyping, while it would be sound. The rationale is that abstraction should only occur when explicitly requested by the user, by ascription. Similarly, there is no floating strengthening rule. This comes from the fact that strengthening is backed in the system and syntax-directed: it happens only when pushing in the typing environment.

Wellformedness We show by an immediate induction that typing a module expression ($\Gamma \vdash^{\diamondsuit} \mathtt{M} : \mathtt{S}$) in a wellformed environment ($\vdash \Gamma : \mathsf{wf}$) produces a wellformed signature: $\Gamma \vdash \mathtt{S} : \mathsf{wf}$

5.4 Resolving signature avoidance by zipper simplification

The type system defined so far only introduces zippers during projection (Rule Typ-M-Proj). From there, zippers are transported by the other typing rules. They may only be eliminated by subtyping at an ascription. Yet, conceptually, zipper contexts are a typechecking tool

to delay signature avoidance, but should be kept only if they are still needed by the zipped signature. As we show in the example of Section 5.2.3, there are situations where we can remove all or part of a zipper context and simplify the signature. In this section, we give a precise meaning to this simplification.

The key intuition is that, as zippers are not present at runtime, we may extend our notion of code-free subtyping to allow removal or rewriting of floating fields as long as it does not affect the (actually present) zipped signature. That is, we want to define a simplifying relation $\Gamma \vdash \mathtt{S} \rightsquigarrow \mathtt{S}'$ as a restricted form of code-free subtyping and add the following rule to the module typing judgment:

$$\textsc{Typ-M-Zip}\quad \frac{\Gamma \vdash \mathtt{M} : \langle\gamma\rangle\,\mathtt{S} \qquad \Gamma \vdash \langle\gamma\rangle\,\mathtt{S} \rightsquigarrow \langle\gamma'\rangle\,\mathtt{S}'}{\Gamma \vdash \mathtt{M} : \langle\gamma'\rangle\,\mathtt{S}'}$$

Simplifying along subtyping ensures that it is always sound. However, code-free subtyping may abstract types fields, and doing so would lead to *over-abstraction*. To show that our simplification never over-abstracts, we introduce *narrow subtyping* $\lessapprox$ that captures this property. While the primary goal of zipper simplification is to eliminate (or reduce) floating fields in inferred types, this is also useful both to print simpler error messages and to speed up typechecking.

In Section 5.4.1, we extend code-free subtyping to support zippers on the right-hand side and define narrow subtyping. In Sections 5.4.2 to 5.4.6, we present a set of (first-order) simplification rules that are special cases of narrow subtyping. In Section 5.4.7, we describe an algorithm that implements those rules. Finally, in Section 5.4.8, we discuss why narrow subtyping and over-abstraction.

5.4.1 Subtyping with zippers and narrow subtyping

Subtyping with zippers

We now extend the definition of subtyping to allow subtyping between signatures containing zippers on both sides by just adding the following rule (keeping all the previous rules, including Rule Sub-S-Zipper):

$$\textsc{Sub-S-ZipZip}\quad \frac{\overline{\mathtt{D}_1}' \subseteq \overline{\mathtt{D}_1} \qquad \Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \overline{\mathtt{D}_1}' \leqslant \overline{\mathtt{D}_2} \qquad \Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \mathtt{S}_1 \lesssim \mathtt{S}_2}{\Gamma \vdash \langle A : \overline{\mathtt{D}_1}\rangle\,\mathtt{S}_1 \lesssim \langle A : \overline{\mathtt{D}_2}\rangle\,\mathtt{S}_2}$$

This rule can be seen as acting on the signatures before the projection, i.e., it commutes with unzipping. However, as floating fields are not present at runtime, we can use the more general coercion subtyping on them: subtyping on floating fields is always code-free!

Narrow subtyping

Code-free subtyping allows for over-abstraction, which we want to prevent, while equivalence prevents the removal of floating fields. We thus define *narrow subtyping* $\lessapprox$ as a *restriction* of code-free subtyping that prevents the loss of visible type equalities by the same set of rules, except for Sub-S-Zipper and Sub-S-Sig which have been replaced by the following variants,

restricted by additional premises:

$$\textsc{Sub-S-SigR}\quad \frac{\Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \overline{\mathtt{D}_1} \mathbin{/\!/} A \lessapprox \overline{\mathtt{D}_2} \qquad \Gamma \uplus A : \overline{\mathtt{D}_2} \vdash \overline{\mathtt{D}_2} \mathbin{/\!/} A \lessapprox \overline{\mathtt{D}_1}}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}_1}\ \mathtt{end} \lessapprox \mathtt{sig}_A\ \overline{\mathtt{D}_2}\ \mathtt{end}}$$

$$\textsc{Sub-S-ZipperR}\quad \frac{\begin{array}{lll}\Gamma_1 = \Gamma \uplus A : \overline{\mathtt{D}_1}, \mathtt{module}\,\mathbb{Z} : \mathtt{S}_1 & \Gamma_2 = \Gamma \uplus A : \overline{\mathtt{D}_2}, \mathtt{module}\,\mathbb{Z} : \mathtt{S}_2, \overline{\mathtt{D}_2}' & \\ \overline{\mathtt{D}_1}' \sqsubset \overline{\mathtt{D}_1} & \Gamma_1 \vdash \overline{\mathtt{D}_1}' \mathbin{/\!/} A \lessapprox \overline{\mathtt{D}_2}, \overline{\mathtt{D}_2}' & \Gamma_1 \vdash \mathtt{S}_1 \lessapprox \mathtt{S}_2 \\ \overline{\mathtt{D}_2}'' \sqsubset \overline{\mathtt{D}_2}, \overline{\mathtt{D}_2}' & \Gamma_2 \vdash \overline{\mathtt{D}_2}'' \mathbin{/\!/} A \lessapprox \overline{\mathtt{D}_1} & \Gamma_2 \vdash \mathtt{S}_1 \mathbin{/\!/} A.\mathbb{Z} \lessapprox \mathtt{S}_2\end{array}}{\Gamma \vdash \langle A : \overline{\mathtt{D}_1} \rangle\, \mathtt{S}_1 \lessapprox \langle A : \overline{\mathtt{D}_2} \rangle\, \mathtt{S}_2}$$

Rule Sub-S-SigR for signatures disallow the removal and reordering of fields and requires field-by-field subtyping in both directions. Rule Sub-S-ZipperR still allows for $\overline{\mathtt{D}_2}$ to contain fewer floating fields than $\overline{\mathtt{D}_1}$ and appearing in a different order as Rule Sub-S-Zipper (premises on the second line). However, it also requires (premises on the last line) the converse subtyping up to the reintroduction of some floating fields $\overline{\mathtt{D}_2}'$ that have been dropped from $\overline{\mathtt{D}_1}$. As will appear in Section 5.4.8, narrow subtyping is somehow the composition of an equivalence – preventing over-abstraction and reversible – with a projection – removing unreachable floating fields and irreversible.

5.4.2 Simplification overview

In the rest of this section, we give the specification of a simplification algorithm that transforms a signature along the narrow subtyping defined in Section 5.4.1, which we can use to reduce the size of the zipper. When we can remove the zipper altogether, it coincides with solving signature avoidance. When some floating fields remain, it delays signature avoidance.

Restrictions Our simplification algorithm does not explore all possibilities of narrow subtyping, but has some restrictions:

1. Given a signature $\langle \gamma \rangle\, \mathtt{S}$ it only tries to remove floating fields from γ, but would not invent new ones (even if it could make for a smaller zipper context).

2. it does not move fields in the zipped signature $\mathtt{S}$ – while code-free subtyping allows it to – but only does a rewriting of the content of the fields, rewiring the type and module equalities.

3. it uses a first-order criterion for applicative functors: it only rewrites functor applications when aliases (for either a lost functor or a lost argument) are available, as does the anchoring of M^{ω} (Section 4.3.2).

Overview At a high level, the simplification follows three elementary rules (*drop*, *move*, *split*), as it tries to simplify a single floating component, and one rule to re-organize zippers (*skip*). In Section 5.4.3, we discuss simplification by dropping a field. In Section 5.4.4, we introduce the simplification by moving a field. In Section 5.4.5, we extend the moving simplification by adding the possibility to split a floating module field. In Section 5.4.6, we show how to treat floating fields that cannot be simplified. Finally, in Section 5.4.7, we discuss technical aspects for an algorithm that would implement those simplification techniques.

5.4.3 Dropping a field

First, if a floating field is useless, i.e., does not appear in the free variables of the signature, we may just remove it, as done by Rule Simp-Drop.

$$\textsc{Simp-Drop}\ \frac{I = \mathsf{dom}(\mathtt{D}) \qquad A.I \notin \mathsf{fv}(\mathtt{S})}{\Gamma \vdash \langle A : \mathtt{D} \rangle\, \mathtt{S} \rightsquigarrow \mathtt{S}}$$

This rule maintains wellformedness, as the identifier does not appear in the free variables. The rule is indeed included in narrow subtyping: the forward subtyping is shown by just ignoring the declaration $\mathtt{D}$ (not putting it in the list of compared declarations $\overline{\mathtt{D}_0}$), while the reverse subtyping is shown by putting $\mathtt{D}$ back (as part of the added declarations $\overline{\mathtt{D}_1}$).

Inlining To simplify fields of zippers, we may use normalization inline floating type definitions and floating module-type definitions. We may then use Simp-Drop to remove them. For a non-abstract type field, where $u \neq A.t$, we have:

$$\langle A : \mathtt{type}\ t = \mathtt{u} \rangle\, \mathtt{S} \lessapprox \langle A : \mathtt{type}\ t = \mathtt{u} \rangle\, \mathtt{S}[A.t \mapsto \mathtt{u}] \gtrapprox \mathtt{S}[A.t \mapsto \mathtt{u}]$$

The first subtyping is just normalization, while the second is Simp-Drop (after the substitution, $A.t$ is no longer in the free variables of $\mathtt{S}[A.t \mapsto u]$).

5.4.4 Moving away a field

However, there are cases where the floating field is not useless (it appears in the signature), but can still be removed. Indeed, there might be a declaration in $\mathtt{S}$ that can take up the same role as the floating field, without loss of type information, as shown in Section 5.2.3. We start by considering a floating *type* field, and we then consider a module field.

Anchoring point

Let us consider a zipper signature $\langle A : \mathtt{D} \rangle\, \mathtt{S}$ where I is the identifier of $\mathtt{D}$. We look for a declaration that we call the *anchoring point* for $A.I$ inside $\mathtt{S}$, which we write $\mathsf{anchor}(\Gamma, A.I, \mathtt{S})$ if it exists. It must validate three conditions:

1. for a type field, it should be of the form $\mathtt{type}\, t' = A.t$. For a module identity, it should be of the form $\mathtt{module}\, X' : (= A.X < \mathtt{S})$ where $\mathtt{S}$ is equivalent to the signature of $A.X$ (not just a supertype).
2. it must be in a strictly positive position, not inside a functor nor a module type.
3. it must come before any other occurrence of $A.I$ (which are called *usage points*).

Formally, we have for a type field (the typing environment argument only serves for floating module fields, we leave it for regularity):

$$\mathsf{anchor}(\Gamma, A.t, \mathtt{S}) = \mathbb{Z}.X_1.(\dots).X_n.u$$

if and only if there exists n self-references $A_0, \dots, A_n$, and n lists of declarations $\overline{\mathtt{D}_1}, \dots, \overline{\mathtt{D}_n}$ and $\overline{\mathtt{D}_1}', \dots, \overline{\mathtt{D}_n}'$, and n signatures $\mathtt{S}_1, \dots, \mathtt{S}_n$ such that:

$$\begin{aligned}
\mathtt{S} &= \mathtt{sig}_{A_0}\ \overline{\mathtt{D}_0}\ ;\ \mathtt{module}\ X_1 : \mathtt{S}_1\ ;\ \overline{\mathtt{D}_0}'\ \mathtt{end} && \wedge A.t \notin \mathsf{fv}(\overline{\mathtt{D}_0}) \\
\mathtt{S}_1 &= \mathtt{sig}_{A_1}\ \overline{\mathtt{D}_1}\ ;\ \mathtt{module}\ X_2 : \mathtt{S}_2\ ;\ \overline{\mathtt{D}_1}'\ \mathtt{end} && \wedge A.t \notin \mathsf{fv}(\overline{\mathtt{D}_1}) \\
&\vdots \\
\mathtt{S}_n &= \mathtt{sig}_{A_n}\ \overline{\mathtt{D}_n}\ ;\ \mathtt{type}\ u = A.t\ ;\ \overline{\mathtt{D}_n}'\ \mathtt{end} && \wedge A.t \notin \mathsf{fv}(\overline{\mathtt{D}_n})
\end{aligned}$$

That is, there is cascade of nested signatures of depth n where $A.t$ is not mentioned until the field $\mathtt{type}\, u = A.t$.

Contextual path substitution

If there exists an anchoring point for $A.t$ at $\mathbb{Z}.X_1.(\dots).X_n.u$, we may remove the floating field by (intuitively) replacing occurrences of $A.I$ by $\mathbb{Z}.X_1.(\dots).X_n.u$. However, the path to access u is not the same everywhere inside the signature. Therefore, we need a special form of substitution, *contextual path substitution* written $\mathtt{S}[A.t \Rightarrow \mathbb{Z}.X_1.(\dots).X_n.u]$, which works as follows:

- we replace $\mathtt{type}\, u = A.t$ by $\mathtt{type}\, u = A_n.u$ (deep in the signature)
- in the declarations $\overline{\mathtt{D}_n}'$, u is accessible by $A_n.u$, so we may substitute $A.t$ by $A_n.u$. There is nothing to substitute in $\overline{\mathtt{D}_n}$, as $A.t$ does not appear free.
- in the declaration $\overline{\mathtt{D}_{n-1}}'$, u is accessible by $A_{n-1}.X_n.u$, so we may substitute $A.t$ by $A_{n-1}.X_n.u$. Again, there is nothing to substitute in $\overline{\mathtt{D}_{n-1}}$.
- ...
- in the declarations $\overline{\mathtt{D}_1}'$, u is accessible by $A_1.X_2.(\dots).X_n.u$, so we may substitute $A.t$ by $A_1.X_2.(\dots).X_n.u$. Again, there is nothing to substitute in $\overline{\mathtt{D}_1}$.
- finally, in the declarations $\overline{\mathtt{D}_0}'$, u is accessible by $A_0.X_1.(\dots).X_n.u$, so we may substitute $A.t$ by $A_0.X_1.(\dots).X_n.u$. There is nothing to substitute in $\overline{\mathtt{D}_0}$.

Basically, when visiting $\mathtt{S}$, it substitutes $A.I$ by $A_1.X_2.(\dots).X_n.u$ stripped of it common prefix with the path of the current point (and rewired at the enclosing self-reference).

Simplification rule

Using anchoring points and contextual path substitution, we have the following rule:

$$\textsc{Simp-Move}\quad \frac{I = \mathsf{dom}(()\mathtt{D}) \qquad \mathsf{anchor}(\Gamma, A.I, \mathtt{S}) = \mathbb{Z}.P.I'}{\Gamma \vdash \langle A : \mathtt{D}\rangle\, \mathtt{S} \rightsquigarrow \mathtt{S}[A.I \Rightarrow \mathbb{Z}.P.I']}$$

Overall, the rule allows to show the narrow subtyping between those two signatures, assuming that $A.t \notin \mathsf{fv}(\overline{\mathtt{D}_0}), \dots, A.t \notin \mathsf{fv}(\overline{\mathtt{D}_n})$:

$\langle A : \mathtt{type}\ t = A.t\rangle$
$\mathtt{sig}_{A_0}$
  $\overline{\mathtt{D}_0}$
  $\mathtt{module}\, X_1 : \mathtt{sig}_{A_1}$
    $\overline{\mathtt{D}_1}$
      $\ddots$
        $\overline{\mathtt{D}_{n-1}}$
        $\mathtt{module}\, X_n : \mathtt{sig}_{A_n}$
          $\overline{\mathtt{D}_n}$;
          $\mathtt{type}\, u = A.t$;
          $\overline{\mathtt{D}_n}'$
        $\mathtt{end}_{A_n}$
        $\overline{\mathtt{D}_{n-1}}'$
      $\ddots$
    $\overline{\mathtt{D}_1}'$
  $\mathtt{end}_{A_1}$;
  $\overline{\mathtt{D}_0}'$
$\mathtt{end}_{A_0}$

$\approx$

$\mathtt{sig}_{A_0}$
  $\overline{\mathtt{D}_0}$;
  $\mathtt{module}\, X_1 : \mathtt{sig}_{A_1}$
    $\overline{\mathtt{D}_1}$;
      $\ddots$
        $\overline{\mathtt{D}_{n-1}}$
        $\mathtt{module}\, X_n : \mathtt{sig}_{A_n}$
          $\overline{\mathtt{D}_n}$;
          $\mathtt{type}\, u = A_n.u$;
          $\overline{\mathtt{D}_n}'[A.t \mapsto A_n.u]$
        $\mathtt{end}_{A_n}$
        $\overline{\mathtt{D}_{n-1}}'[A.t \mapsto A_{n-1}.X_n.u]$
      $\ddots$
    $\overline{\mathtt{D}_1}'[A.t \mapsto A_1.X_2.(\dots).X_n.u]$
  $\mathtt{end}_{A_1}$;
  $\overline{\mathtt{D}_0}'[A.t \mapsto A_0.X_1.X_2.(\dots).X_n.u]$
$\mathtt{end}_{A_0}$

The fact that this rule maintains wellformedness crucially relies on $A.t$ not being in the free variables of $\overline{\mathtt{D}_0}, \cdots, \overline{\mathtt{D}_n}$. To show the narrow subtyping, the forward direction $\lesssim$ is easy by normalization and abstraction. The reverse direction $\gtrsim$ requires to invent a new floating field, and we choose $\mathtt{type}\, t = A.\mathbb{Z}.X_1.X_2.(\dots).X_n.u$. We conclude by normalization and reflexivity.

Module fields

For a floating module field $\langle A : \texttt{module}\ X : \mathtt{S} \rangle$, the definition of the anchoring point is the same, except that it must be a module declaration with a transparent signature:

$$\mathtt{S}_n = \mathtt{sig}_{A_n}\ \overline{\mathtt{D}_n}\ ;\ \texttt{module}\ X' : (= A.X < \mathtt{S}')\ ;\ \overline{\mathtt{D}_n'}\ \texttt{end}\ \wedge\ A.X \notin \mathsf{fv}(\overline{\mathtt{D}_n})$$

Besides, we have an additional equivalence condition:

$$\Gamma \uplus A_0 : \overline{\mathtt{D}_0} \uplus A_1 : \overline{\mathtt{D}_1} \uplus \ldots \uplus A_n : \overline{\mathtt{D}_n} \vdash \mathtt{S} \approx (\mathtt{S}' \,/\, A.X)$$

Overall, the rule allows to show the subtyping between those two signatures, assuming that $A.t \notin \mathsf{fv}(\overline{\mathtt{D}_0}), \ldots, A.t \notin \mathsf{fv}(\overline{\mathtt{D}_n})$:

$$\begin{array}{l}\langle A : \texttt{module}\ X : \mathtt{S} \rangle \\ \mathtt{sig}_{A_0} \\ \quad \overline{\mathtt{D}_0} \\ \quad\quad \ddots \\ \quad\quad\quad \overline{\mathtt{D}_n} \\ \quad\quad\quad \texttt{module}\ X' : (= A.X < \mathtt{S}') \\ \quad\quad\quad \overline{\mathtt{D}_n'} \\ \quad\quad \ddots \\ \quad \overline{\mathtt{D}_0'} \\ \mathtt{end}_{A_0}\end{array} \quad \approx \quad \begin{array}{l}\mathtt{sig}_{A_0} \\ \quad \overline{\mathtt{D}_0} \\ \quad\quad \ddots \\ \quad\quad\quad \overline{\mathtt{D}_n} \\ \quad\quad\quad \texttt{module}\ X' : \mathtt{S} \\ \quad\quad\quad \overline{\mathtt{D}_n'}[A.X \mapsto A_n.X'] \\ \quad\quad \ddots \\ \quad \overline{\mathtt{D}_0'}[A.X \mapsto A_0.X_1.(\ldots).X'] \\ \mathtt{end}_{A_0}\end{array}$$

Again, wellformedness relies on $A.X$ not being in the free variables of $\overline{\mathtt{D}_0}, \cdots, \overline{\mathtt{D}_n}$. To show narrow subtyping, the forward direction $\lesssim$ uses reflexivity and normalization. For the anchoring point, we use Rule Sub-S-LooseAlias and the equivalence condition. The reverse direction $\gtrsim$ requires to invent a new floating field, and we choose:

$$\texttt{module}\ X : (= A.\mathbb{Z}.X_1.(\ldots).X_n.X' < \mathtt{S})$$

We again use the equivalence condition to conclude.

5.4.5 Splitting a field

While powerful, the simplification by moving fields might not be enough. We have the following example:

Example 5.4.1 (Splitting a module)**.** *We consider the following zipper signature:*

$$\begin{array}{l}\langle A_1 : \texttt{module}\ X : \mathtt{sig}_A\ \texttt{type}\ t = A.t\ \texttt{end} \rangle \\ \quad \mathtt{sig}_B\ \texttt{type}\ u = A_1.X.t \\ \quad\quad\quad \texttt{module}\ X' : (= A_1.X < \mathtt{sig}_A\ \texttt{type}\ t = A.t\ \texttt{end})\ \texttt{end}\end{array}$$

We cannot drop the floating module field as $A_1.X$ appears in the signature. Second, trying to apply Simp-move *to the field would also fail, as the type $A_1.X.t$ is used before an anchoring point for X. Yet, there is a zipper-less equivalent signature:*

$$\mathtt{sig}_B\ \texttt{type}\ u = B.u\ \texttt{module}\ X' : \mathtt{sig}_C\ \texttt{type}\ t = B.u\ \texttt{end}\ \texttt{end}$$

We can obtain this by first introducing *a* temporary *floating field, and then only* moving *fields:*

$$\begin{array}{ll}
& \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end}\rangle \qquad \text{(Initial signature)}\\
& \quad \mathtt{sig}_B\ \mathtt{type}\, u = A_1.X.t\\
& \qquad \mathtt{module}\, X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end}\\
& \langle A_0 : \mathtt{type}\ t = A_0.t\rangle\, \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\rangle \qquad \text{(Adding a field)}\\
& \quad \mathtt{sig}_B\ \mathtt{type}\, u = A_1.X.t\\
& \qquad \mathtt{module}\, X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end}\\
\lessapprox & \langle A_0 : \mathtt{type}\ t = A_0.t\rangle\, \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\rangle \qquad \text{(Normalization)}\\
& \quad \mathtt{sig}_B\ \mathtt{type}\, u = A_0.t\\
& \qquad \mathtt{module}\, X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end}\\
\lessapprox & \langle A_0 : \mathtt{type}\ t = A_0.t\rangle \qquad \text{(Moving away the module field)}\\
& \quad \mathtt{sig}_B\ \mathtt{type}\, u = A_0.t\\
& \qquad \mathtt{module}\, X' : \mathtt{sig}_C\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\ \mathtt{end}\\
\lessapprox & \mathtt{sig}_B\ \mathtt{type}\, u = B.u \qquad \text{(Moving away the type field)}\\
& \quad \mathtt{module}\, X' : \mathtt{sig}_C\ \mathtt{type}\ t = B.u\ \mathtt{end}\ \mathtt{end}
\end{array}$$

Simplification rule The splitting rule Simp-Split is meant to be used in conjunction with the rules for moving and dropping fields, and actually fits quite easily in our setting. We simply allow to *temporarily* introduce new floating fields if and only if those additional fields help the module field get simplified, but get removed in the end. For type field, we have:

$$\textsc{Simp-Split-Type}\quad \frac{\mathtt{S}_0 = \mathtt{sig}_B\ \overline{\mathtt{D}}\ \mathtt{end} \qquad \mathtt{type}\, t = B.t \in \overline{\mathtt{D}} \qquad \Gamma \vdash \langle A_0 : \mathtt{type}\ t = A_0.t\,;\, A : \mathtt{module}\ X : \mathtt{S}_0[B.t \mapsto A_0.t]\rangle\, \mathtt{S} \rightsquigarrow \tilde{\mathtt{S}}}{\Gamma \vdash \langle A : \mathtt{module}\ X : \mathtt{S}_0\rangle\, \mathtt{S} \rightsquigarrow \tilde{\mathtt{S}}}$$

Importantly, this rule applies only if the module X as a structural signature, not a functor signature. Besides, anchoring points are never inside functors themselves. This is where the *first-order* restriction can be seen, as we do not try to *split* individual fields of functors. Simplification of functors can only use Rule Simp-Move-Mod. The rule also pattern matches on the *absence* of a zipper on the right-hand side signature, therefore only allowing splitting when it helps simplification.

For modules, we have:

$$\textsc{Simp-Split-Mod}\quad \frac{\mathtt{S}_0 = \mathtt{sig}_B\ \overline{\mathtt{D}_0}\,;\ \mathtt{module}\ X_1 : \mathtt{S}_1\,;\ \overline{\mathtt{D}_0}'\ \mathtt{end} \qquad \Gamma \vdash \mathtt{S}_1 : \mathtt{wf} \qquad \mathtt{S}_0' = \mathtt{sig}_B\ \overline{\mathtt{D}_0}\,;\ \mathtt{module}\ X_1 : (= A_1.X_1 < \mathtt{S}_1)\,;\ \overline{\mathtt{D}_0}'\ \mathtt{end} \qquad \Gamma \vdash \langle (A_1 : \mathtt{module}\ X_1 : \mathtt{S}_1)\,;\, (A : \mathtt{module}\ X : \mathtt{S}_0')\rangle\, \mathtt{S} \rightsquigarrow \tilde{\mathtt{S}}}{\Gamma \vdash \langle A : \mathtt{module}\ X : \mathtt{S}_0\rangle\, \mathtt{S} \rightsquigarrow \tilde{\mathtt{S}}}$$

As for types, we introduce a new floating field to allow $A.X$ and $A.X.X_1$ to be simplified separately.

5.4.6 Skipping a field

The three simplification rules pattern-match on the right-most floating field. If the right-most field does not fall into one of the three cases above, we can skip it and leave it as a floating component. It just amounts to consider the last field $\mathtt{D}$ as part of the visible signature.

$$\textsc{Simp-Skip}\quad \frac{\mathsf{dom}(\mathtt{D}) = I \qquad \Gamma \vdash \left\langle A : \overline{\mathtt{D}_1}\right\rangle ((\mathtt{sig}_B\ \mathtt{D}; \mathtt{module}\ \mathbb{Z} : \mathtt{S}\ \mathtt{end})[A.I \mapsto B.I]) \rightsquigarrow \left\langle A : \overline{\mathtt{D}_1}'\right\rangle \mathtt{sig}_B\ \mathtt{D}'; \mathtt{module}\ \mathbb{Z} : \mathtt{S}'\ \mathtt{end}}{\Gamma \vdash \left\langle A : \overline{\mathtt{D}_1}; \mathtt{D}\right\rangle \mathtt{S} \rightsquigarrow \left\langle A : \overline{\mathtt{D}_1}'; \mathtt{D}'\right\rangle \mathtt{S}'[B.I \mapsto A.I]}$$

5.4.7 Simplification algorithm

From there, the main challenge is to do all of simplifications presented in the last subsections as efficiently as possible. In this subsection we consider the simplification of a signature of the form:

$$\langle A : \mathtt{D}_1; \ldots; \mathtt{D}_n \rangle\, \mathtt{S}$$

Each declaration $\mathtt{D}_k$ contains the identifier I_k. We present an algorithm that works in three steps. First, *scanning* browses the zipper and the signature to collect information about the usage points of each floating field. Then the *constraint resolution* computes which floating field are going to be *dropped*, *moved*, *split*, or *skipped*. Finally, a *simplification* pass revisits the signature to apply the corresponding transformations. We have not implemented this algorithm yet, but present its proposed structure.

Scanning

We do the first pass with a mutable map θ matching each floating identifier $A.I_k$ with a list of *usage points* stored in order of appearance. Each usage point is one of three kinds:

- **zipper** $(A.I_\ell)$ indicates that $A.I_k$ is used in the floating field $A.\mathtt{D}_\ell$.
- **anchor** $(A.I_k.\overline{X}.I,\ \mathbb{Z}.\overline{X}'.I')$ is used when the declaration at $\mathbb{Z}.\overline{X}'.I'$ could serve as an anchoring point for the subfield $A.I_k.\overline{X}.I$ if no occurrence of $A.I_k$ appears before $\mathbb{Z}.\overline{X}'.I'$.
- **usage** is used otherwise. For module fields where $A.I_k$ is used as a prefix, we store the whole path, as **usage** $(A.I_k.\overline{X}.I)$; otherwise, **usage** has no argument.

We then define a function $\mathsf{visit}_P(\cdot)$ that visits the zipper and the signature recursively, in order of appearance, while updating the map (hence adding new elements to the tail of the list). P is an optional argument that is only be passed when visiting the zipper body $\mathtt{S}$. Initially, θ maps each I_k to an empty list. When visiting the zipper context, only **zipper** $(\cdot)$ usage points are used.

When visiting the signature $\mathtt{S}$, the optional path argument P indicates the path to the root $\mathbb{Z}$ if still accessible, or none otherwise. The initial call is $\mathsf{visit}_{\mathbb{Z}}\,\mathtt{S}$. A type declaration is first check as a possible anchoring point when the path is nonempty. If not an anchoring point or if the path is empty, then it is a usage point. The path is set to none for recursive calls when entering, (1) a functor or a module-type, (2) a submodule with a transparent signature, as paths reaching inside the submodule are normalized away, and (3) a submodule with a module-type signature (as normalization should be used first). The initial call is $\mathsf{visit}_{\mathbb{Z}}\,\mathtt{S}$.

Constraint resolution

For constraint resolution, we use the information collected in the first pass to compute the set of simplifications that can be applied to the signature. For that purpose, we introduce a single assignment map Ω from paths of the form $A.I_k.\overline{X}.I'$ to actions, initially undefined everywhere, and which may be set once one value among **drop**, **move**, **split**, and **skip**. Constraint resolution updates Ω and returns a list of substitutions Θ to be applied to $\mathtt{S}$.

We visit the floating fields I_k in reverse order, i.e., for k ranging from n to 1. We consider the list θI_k of usage points collected in the first phase. We first remove from θI_k all usage points **zipper** $(A.I_\ell)$ for which $\Omega(I_\ell)$ is **drop**, **move**, or **split**, since then I_ℓ will be removed during the simplification. We then scan the list θI_k of remaining usage points in order:

- if θI_k is empty, we set $\Omega(I_k)$ to **drop**.
- if the head of θI_k is **anchor** $(P,\ P')$ we set $\Omega(I_k)$ to **move** and Θ to $[P \Rightarrow P'] \circ \Theta$.

- Otherwise, the head of θI_k is a usage point. If the field I_k is a module declaration, we try to split that field. That is, we consider a local assignment map ω for suffixes of $A.X$ and we go through the list θI_k of usage points, with the following cases:
  - **usage** $(A.I_k.\overline{X}.I)$: if $A.I_k.\overline{X}.I$ or any prefix of the form $A.I_k.\overline{X}'$ is already set to **move** in ω, we remove the current usage point from θI_k, and continue with the rest of θI_k; otherwise, we set $\Omega(I_k)$ to **skip**, and proceed with the successor of I_k (discarding ω).
  - **anchor** (P, P') : if no prefix of P is already set to **move**, we set $\omega(A.I_k.\overline{X}.I)$ to **move** and Θ to $[P \Rightarrow P'] \circ \Theta$.

  If all usage points of the modules have been dealt with, we set $\Omega(I_k)$ to **split** and discard ω.

Simplification

Finally, we return the zipper $\left\langle A : \mathtt{D}_k{}^{k\in 1..n \wedge \Omega(A.I_k)=\mathbf{skip}} \right\rangle (\mathtt{S}\Theta)$ whose context just retained the floating fields set to **skip**, applying the path contextual substitution Θ to $\mathtt{S}$.

5.4.8 Preservation of typability

In this section we show that narrow subtyping (which includes simplification) never prevents further typings, i.e., if a typing derivation of $\mathtt{M}$ inferred a type $\mathtt{S}$ for a subexpression N that could be simplified into $\mathtt{S}'$, then we could have continued the typing derivation by taking $\mathtt{S}'$ instead of $\mathtt{S}$ for the type of N without loosing information (up to further simplifications). The detailed proof is not included here, we only sketch its structure. Informally, we could say:

$$\text{If } \Gamma \vdash \mathtt{M} \left[\Gamma' \vdash N : \mathtt{S}\right] : \mathtt{S}_0 \text{ and } \Gamma' \vdash \mathtt{S} \lessapprox \mathtt{S}', \text{ then } \Gamma \vdash \mathtt{M} \left[\Gamma' \vdash N : \mathtt{S}'\right] : \mathtt{S}'_0 \text{ and } \Gamma \vdash \mathtt{S}_0 \lessapprox \mathtt{S}'_0.$$

We prove an almost equivalent, simpler, *principality* property: *If* $\Gamma \vdash \mathtt{M} : \mathtt{S}$ *is derivable (with or without simplifications) and* Γ' *is a typing environment where simplifications have been performed, which we write* $\vdash \Gamma \lessapprox \Gamma'$*, then we have* $\Gamma' \vdash \mathtt{M} : \mathtt{S}'$ *for some* $\mathtt{S}'$ *that is equivalent to a simplification of* $\mathtt{S}$.

We do so by establishing a correspondence between Γ and Γ', keeping track of floating fields that have been simplified, instead marked as *eventually removable* fields. This enables a dual view, where removable fields can either be seen or ignored.

Technically, we introduce ZipML^F, a conservative extension of ZipML with *full* zippers $\left\langle A : \overline{\mathtt{D}} \mid \overline{\mathtt{D}}' \right\rangle \mathtt{S}$. While a simple zipper $\langle A : \mathtt{D} \rangle\, \mathtt{S}$ stands, intuitively, for a signature $\mathtt{sig}_A\ \overline{\mathtt{D}}, \mathbb{Z} : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}$ that has been projected on field $\mathbb{Z}$, keeping the left floating fields $\overline{\mathtt{D}}$ but forgetting about the right floating fields $\overline{\mathtt{D}}'$ that follow the signature $\mathtt{S}$ of field $\mathbb{Z}$, a full zipper also retains the right floating fields, which may then refer to all other floating fields via the label A and to the signature $\mathtt{S}$ via the path $A.\mathbb{Z}$. Hence, a full zipper $\left\langle A : \overline{\mathtt{D}} \mid \overline{\mathtt{D}}' \right\rangle \mathtt{S}$ is well-formed whenever the signature $\mathtt{sig}_A\ \overline{\mathtt{D}}, \mathbb{Z} : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}$ is. A simple zipper $\langle A : \mathtt{D} \rangle\, \mathtt{S}$ is then just a particular case $\langle A : \mathtt{D} \mid \varnothing \rangle\, \mathtt{S}$ of a full zipper with an empty sequence of right floating fields.

Strengthening is extended to full zippers in the obvious way, while most definitions treat zippers abstractly and need not be redefined, [9] except for subtyping: we replace Rule Sub-S-Zipper (which will be derivable), by the following Rule Sub-S-ZipperF that allows subtyping between full zippers:

$$\textsc{Sub-S-ZipperF}\quad \frac{\Gamma_1 = \Gamma \uplus A : (\overline{\mathtt{D}_1}, \mathbb{Z} : \mathtt{S}, \overline{\mathtt{D}_1}') \qquad \overline{\mathtt{D}_2}'' \sqsubset \overline{\mathtt{D}_1}, \overline{\mathtt{D}_1}' \qquad \Gamma_1 \vdash \overline{\mathtt{D}_2}'' \mathbin{/\!/} A < \overline{\mathtt{D}_2}, \overline{\mathtt{D}_2}' \qquad \Gamma_1 \vdash \mathtt{S}_1 < \mathtt{S}}{\Gamma \vdash \left\langle A : \overline{\mathtt{D}_1} \mid \overline{\mathtt{D}_1}' \right\rangle \mathtt{S}_1 < \left\langle A : \overline{\mathtt{D}_2} \mid \overline{\mathtt{D}_2}' \right\rangle \mathtt{S}}$$

[9] Definition should respect the unzipped signature view of full zippers.

Interestingly, dropping right floating fields is possible as a particular case of subtyping. In particular, $\Gamma \vdash \langle A : \overline{\mathtt{D}} \mid \overline{\mathtt{D}}' \rangle\, \mathtt{S} < \langle A : \overline{\mathtt{D}} \rangle\, \mathtt{S}$ always holds. We define the *projection* of a signature $\mathtt{S}$, and write $\lfloor \mathtt{S} \rfloor$ for the signature obtained by the removal of all right floating fields of $\mathtt{S}$ (including deep zippers). While toplevel projection preserves well-formedness, deeper projection may not—as is usually the case with subtyping.Indeed, while left floating fields of a signature cannot access its own right floating fields, it can access the right floating fields of another signature in scope. We say that a signature is *projectable* if its projection is well-formed. In particular, it means that the right floating fields of its zippers are not accessed by the rest of the signature (but can be accessed by other right floating fields).

The key point is that narrow subtyping in ZipML can be seen in ZipML^{F}as an equivalence $\approx$ followed by a projection. Therefore, to show that narrow subtyping preserves typing, we *delay* the projection in ZipML^F, keeping the projected fields in the right-hand side. This enables reasoning by equivalences in ZipML^F, and, as long as we preserved projectability, project the judgments back into ZipML at the end.In fact, the relation $\Gamma \vdash \mathtt{S} \lessapprox \mathtt{S}'$ with simple zippers in ZipML has been defined to coincide with finding an equivalent signature $\mathtt{S}''$ in ZipML^F(which usually requires full zippers) whose projection is $\mathtt{S}'$. That is, we have $\Gamma \vdash \mathtt{S} \lessapprox \mathtt{S}'$ in ZipML is equivalent in ZipML^{F}to the existence of $\mathtt{S}''$ whose projection is $\mathtt{S}'$ so that $\Gamma \vdash \mathtt{S} \approx \mathtt{S}''$.

Technically, we link a typing environment Γ of ZipML (without full zippers) with an *equivalent* typing environment Γ' of ZipML^F, where some signatures have been rewritten along the equivalence of ZipML^F. We write $\Gamma \vdash \mathtt{S} \overset{\triangleright}{\approx} \mathtt{S}'$ to mean $\Gamma \vdash \mathtt{S} \approx \mathtt{S}'$ in ZipML^{F}when $\mathtt{S}$ is in ZipML and $\mathtt{S}'$ is projectable. If Γ is in ZipML, this coincides with narrow subtyping. We extend the notation $\overset{\triangleright}{\approx}$ to declarations in the obvious way. We then extend it typing environments with the following rule:

$$\frac{\vdash \Gamma \overset{\triangleright}{\approx} \Gamma' \qquad \Gamma' \vdash \overline{\mathtt{D}} \overset{\triangleright}{\approx} \overline{\mathtt{D}}' \qquad \lfloor \Gamma' \rfloor \vdash \lfloor \overline{\mathtt{D}}' \rfloor : \mathsf{wf}}{\vdash \Gamma \uplus A : \overline{\mathtt{D}} \overset{\triangleright}{\approx} \Gamma' \uplus A : \overline{\mathtt{D}}'}$$

By construction, $\vdash \Gamma \overset{\triangleright}{\approx} \Gamma'$ implies that Γ is in ZipML. The premise $\lfloor \Gamma' \rfloor \vdash \lfloor \overline{\mathtt{D}}' \rfloor : \mathsf{wf}$ ensures that $\lfloor \Gamma' \uplus A : \mathtt{D}' \rfloor$ is well-formed.

To establish a tight correspondence between ZipML and ZipML^F, we must not only project signatures but also judgments—and derivations. We write $\Gamma' \lfloor \vdash \rfloor\, \mathtt{S}$ to mean that both $\Gamma' \vdash \mathtt{S} : \mathsf{wf}$ holds in ZipML^{F}and $\lfloor \Gamma' \rfloor \vdash \lfloor \mathtt{S} \rfloor : \mathsf{wf}$ holds in ZipML[10]. We similarly define $\Gamma' \lfloor \vdash \rfloor\, \mathtt{M} : \mathtt{S}$ and $\Gamma' \lfloor \vdash \rfloor\, \mathtt{S} < \mathtt{S}'$ The principality property, stated at the beginning of this section, is the projection in ZipML of the following lemma, stated in ZipML^F:

Lemma 9. *If* $\Gamma \vdash \mathtt{M} : \mathtt{S}$ *and* $\vdash \Gamma \overset{\triangleright}{\approx} \Gamma'$*, then there exists* $\mathtt{S}'$ *such that* $\Gamma' \lfloor \vdash \rfloor\, \mathtt{M} : \mathtt{S}'$ *and* $\Gamma' \vdash \mathtt{S} \overset{\triangleright}{\approx} \mathtt{S}'$

Its proof is a combination of reasoning along equivalences, relying on general metatheoretical properties of ZipML^F, followed by more specific, but easy results about projectability, ensuring that not only judgments can be projected, but their derivation can be done in ZipML.

Simplifications in ZipML^F So far, we have not changed the simplification rule Typ-M-Simpl. Hence, we cannot do more simplifications in ZipML^{F}that in ZipML, although it would be safe to do so. In particular, any simplification along the code-free equivalence is safe. Doing this, simplifications in ZipML^{F}could keep removable floating fields as right floating fields, preparing and enabling their eventual removal later on just by projection.

[10]That is, the derivation can be conducted without using full zippers.

5.5 Properties

In this section we state and prove the soundness of ZipML by elaboration in M^ω. This does not cover zipper simplification. We conjecture the completeness of ZipML with respect to M^ω by sketching a translation from M^ω to ZipML.

5.5.1 Soundness of ZipML by Elaboration in M^ω

The goal is to show that every module expression that typechecks in ZipML without simplification also typechecks in M^ω (as they have the same source language):

$$\vdash^\Diamond \mathtt{M} : \mathtt{S} \quad \Longrightarrow \quad \overset{\omega}{\vdash} \mathtt{M} : \exists^\Diamond \overline{\alpha}.\mathcal{C} \qquad \text{(For some } \overline{\alpha}, \mathcal{C})$$

To distinguish between ZipML and M^ω judgments, we write $\overset{\omega}{\vdash}$ for judgments in M^ω and add a light red background. We consider M^ω with module identities, as presented in Section 4.2.2. To prove this soundness result by induction on the typing derivations we need to generalize it to a non-empty typing environment and link the two output signatures. To that aim, we introduce an elaboration of source typing environments Γ into M^ω ones, written $\overset{\omega}{\vdash} \Gamma : \Gamma^\omega$. Using it, the induction hypothesis for the soundness statement becomes:

$$\Gamma \vdash^\Diamond \mathtt{M} : \mathtt{S} \ \wedge \ \overset{\omega}{\vdash} \Gamma : \Gamma^\omega \quad \Longrightarrow \quad \exists \overline{\alpha}, \mathcal{C}.\ \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge \ \Gamma^\omega \overset{\omega}{\vdash} \mathtt{M} : \exists^\Diamond \overline{\alpha}.\mathcal{C} \tag{5.1}$$

Yet, we need to extend M^ω elaboration of signatures to support zippers for this statement to make sense. The rest of this section is composed as follows:

- we first explain the treatment of zippers
- we discuss the elaboration of abstract types in the environment and extend the induction hypothesis
- we state the elaboration of strengthening, normalization, subtyping and resolution
- we state the elaboration of path typing, signature typing and module typing
- we prove the soundness result

Treatment of zippers

The first difficulty in the soundness statement is that the language of inferred signatures of ZipML is larger than the source signatures of M^ω, with the introduction of zippers signatures $\langle\gamma\rangle\,\mathtt{S}$ and zipper-context accesses $P.A$. Intuitively, zippers are removed at runtime, and could be removed in M^ω. However, to establish a one-to-one correspondence on inferred signatures, we need to keep zippers in both inferred signatures and the environment also in M^ω. Therefore, we define M^ω_{zip}, an extension of M^ω with zipper signatures and zipper context access. We write γ^ω for M^ω zipper contexts. We add the following two rules:

$$\frac{\Gamma \overset{\omega}{\vdash}_A \mathtt{D} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad \Gamma, A : \overline{\mathcal{D}} \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \overset{\omega}{\vdash} \langle A : \mathtt{D}\rangle\,\mathtt{S} : \lambda\overline{\alpha},\overline{\beta}.\langle A : \overline{\mathcal{D}}\rangle\,\mathcal{C}}\ \textsc{MZip-Typ-S-Zip} \qquad \frac{\Gamma \overset{\omega}{\vdash} P : \langle A : \overline{\mathcal{D}}\rangle\,\mathcal{C}}{\Gamma \overset{\omega}{\vdash} P.A : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}\ \textsc{MZip-Typ-P-Zip}$$

We also extend wellformedness and subtyping accordingly. Following the ZipML convention, we use $\tilde{\mathcal{C}}$ to indicate a M^ω signature without zipper. However, and this is a key point, we do not need to redo the soundness of M^ω_{zip}. We only use M^ω_{zip} to have strong-enough induction hypothesis to link ZipML and M^ω signatures. In a typing derivation of M^ω_{zip} that does not use MZip-Typ-S-Zip nor MZip-Typ-P-Zip, we can erase all zippers and obtain a normal derivation

of M^ω. Therefore, we extend the induction hypothesis to:

$$\left.\begin{array}{l}\Gamma \vdash^{\Diamond} \mathtt{M} : \mathtt{S} \\ \overset{\omega}{\vdash} \Gamma : \Gamma^\omega\end{array}\right\} \implies \exists \overline{\alpha}, \mathcal{C}. \left\{\begin{array}{l}\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}\ (\mathbf{1}) \\ \Gamma^\omega \overset{\omega}{\vdash} \mathtt{M} : \exists^{\Diamond}\overline{\alpha}.\mathcal{C}\ (\mathbf{2}) \\ (2) \text{ can be zip-erased}\end{array}\right.$$

Elaboration of environments

We now move our attention to the definition of $\overset{\omega}{\vdash} \Gamma : \Gamma^\omega$, the elaboration of typing environments. A technical point arises from the representation of abstract types as self-referring signatures in ZipML typing environments, as opposed to the existential type variables of M^ω. For instance, consider an environment extended with a type declaration: $\Gamma; (A : \mathtt{type}\, t = A.t)$. We want a rule of the form:

$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega}{\overset{\omega}{\vdash} \Gamma, (A : \mathtt{type}\, t = A.t) : \Gamma^\omega, \alpha, A : \mathtt{type}\, t = \alpha}$$

Yet, we cannot just elaborate $\mathtt{type}\, t = A.t$ in Γ^ω, as it ill-formed: Γ^ω does not contain a binding for $A.t$ (yet). The solution is to use a rule that captures the property that bindings of Γ are self-referring:

$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \qquad \Gamma^\omega, \alpha, A : \mathtt{type}\, t = \alpha \overset{\omega}{\vdash}_A \mathtt{type}\, t = A.t : \mathtt{type}\, t = \alpha}{\overset{\omega}{\vdash} \Gamma, (A : \mathtt{type}\, t = A.t) : \Gamma^\omega, \alpha, A : \mathtt{type}\, t = \alpha}$$

In the environment $\Gamma^\omega, \alpha, A : \mathtt{type}\, t = \alpha$, the declaration $\mathtt{type}\, t = A.t$ is wellformed and elaborates to $\mathtt{type}\, t = \alpha$. Generalized to any declaration, we get the following rule:

M-Typ-E-Decl
$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \qquad \Gamma, \overline{\alpha}, A : \mathcal{D} \overset{\omega}{\vdash}_A \mathtt{D} : \mathcal{D}}{\overset{\omega}{\vdash} (\Gamma, A : \mathtt{D}) : (\Gamma^\omega, \overline{\alpha}, A : \mathcal{D})}$$

This is not an algorithmic rule, as it requires to *guess* the M^ω-declaration $\mathcal{D}$ and the set of abstract types $\overline{\alpha}$. However, during soundness proof, it is easy to guess them, as the context $\Gamma, A : \mathtt{D}$ is the result of a context extension $\Gamma \uplus A : \mathtt{D}_0$. Elaborating $\mathtt{D}_0$ gives us $\overline{\alpha}$ and $\mathcal{D}$. We have the following derived rule:

$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \qquad \Gamma^\omega \overset{\omega}{\vdash}_A \mathtt{D}_0 : \lambda\overline{\alpha}.\mathcal{D}}{\overset{\omega}{\vdash} (\Gamma \uplus A : \mathtt{D}_0) : (\Gamma^\omega, \overline{\alpha}, A : \mathcal{D})}$$

The other rules follow the same intuition:

M-Typ-E-Nil
$$\overset{\omega}{\vdash} \varnothing : \varnothing$$

M-Typ-E-Param
$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \qquad \Gamma^\omega, \overline{\alpha}, Y : \mathcal{C} \overset{\omega}{\vdash} \mathtt{S} : \mathcal{C}}{\overset{\omega}{\vdash} (\Gamma, Y : \mathtt{S}) : (\Gamma^\omega, \overline{\alpha}, Y : \mathcal{C})}$$

M-Typ-E-Zip
$$\frac{\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \qquad \Gamma^\omega, \overline{\alpha}, \gamma^\omega \overset{\omega}{\vdash} \gamma : \gamma^\omega}{\overset{\omega}{\vdash} (\Gamma, \gamma) : (\Gamma^\omega, \overline{\alpha}, \gamma^\omega)}$$

Strengthening

Lemma 10 (Elaboration of strengthening)**.** *Given a path P and a signature $\mathtt{S}$, such that $\Gamma^\omega \overset{\omega}{\vdash} P : \mathcal{C}'$ (**1**) and $\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$ (**2**) and $\Gamma^\omega \overset{\omega}{\vdash} \mathcal{C}' < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$ (**3**), we have*

$$\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} \,/\!/\, P : (\lambda\overline{\alpha}.\mathcal{C})\,\overline{\tau}\ (\mathbf{4})$$

Proof. We proceed by induction on $\mathtt{S}$. We have four cases, depending on the head constructor.

- $(= P' < \mathtt{S}')$ We have $(= P' < \mathtt{S}') \,/\!/\, P = (= P' < \mathtt{S}')$. We know that (2) must have been obtained by M-Typ-Sig-Trans, and therefore the set of $\overline{\alpha}$ must be empty (a transparent signature does not bind new abstract types). That is, (2) is:

  $$\Gamma^\omega \overset{\omega}{\vdash} (= P' < \mathtt{S}') : \mathcal{C}$$

  We conclude by type equivalence (which includes β-reduction): $(\lambda\varnothing.\mathcal{C})\,\varnothing \equiv \mathcal{C}$

- The zipper cases are easy by induction.

- in the other cases of non-zipped signatures, we have $\mathtt{S} \,/\!/\, P = (= P < \mathtt{S})$. To show (4), we use M-Typ-Sig-Trans:

  $$\text{M-Typ-Sig-Trans}\quad \frac{\Gamma \overset{\omega}{\vdash} P : \mathcal{C}' \qquad \Gamma \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \overset{\omega}{\vdash} \mathcal{C}' < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\omega}{\vdash} (= P < \mathtt{S}) : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

  The result of the rule is indeed the same as (4) by type equivalence, while the premises are exactly (1), (2), (3).

□

Elaboration of judgments

As path typing, subtyping, and normalization are mutually recursive, we state four combined properties that are proven by mutual induction. Type and module-type definitions are kept as such and only inlined on demand in ZipML, while they are immediately inlined in M^ω. Therefore, ZipML normalization becomes the identity in M^ω.

Lemma 11 (Elaboration of normalization)**.** *If $\mathtt{S}$ normalizes to $\mathtt{S}'$, i.e., $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$, and if we have $\overset{\omega}{\vdash} \Gamma : \Gamma^\omega$, then both signatures have the same elaboration in M^ω. Namely:*

$$\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \implies \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}.\mathcal{C}$$

Lemma 12 (Elaboration of path-typing)**.** *If $\Gamma \vdash P : \mathtt{S}$ and $\overset{\omega}{\vdash} \Gamma : \Gamma^\omega$, we have:*

$$\Gamma^\omega \overset{\omega}{\vdash} P : (\tau, \mathcal{C}) \;\wedge\; \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : (\tau, \mathcal{C})$$

Lemma 13 (Elaboration of path resolution)**.** *Assuming $\overset{\omega}{\vdash} \Gamma : \Gamma^\omega$, we have both*

- *If $\Gamma \vdash P \triangleright \mathtt{S}$ then $\mathtt{S}$ has the same signature as P but with a fresh identity:*

  $$\Gamma^\omega \overset{\omega}{\vdash} P : (\tau, \mathcal{C}) \;\wedge\; \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\alpha.(\alpha, \mathcal{C})$$

- *If $\Gamma \vdash P \triangleright P'$ then P and P' have the same identity tag, and P' has a better signature:*

  $$\Gamma^\omega \overset{\omega}{\vdash} P : (\tau, \mathcal{C}) \;\wedge\; \Gamma^\omega \overset{\omega}{\vdash} P' : (\tau, \mathcal{C}') \;\wedge\; \Gamma^\omega \overset{\omega}{\vdash} \mathcal{C}' < \mathcal{C}$$

Lemma 14 (Elaboration of subtyping)**.** *The elaboration preserves subtyping relationship: If $\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}'.\mathcal{C}'$ then $\Gamma \vdash \mathtt{S} \leq \mathtt{S}'$ implies $\Gamma^\omega \overset{\omega}{\vdash} \lambda\overline{\alpha}.\mathcal{C} < \lambda\overline{\alpha}'.\mathcal{C}'$. This does not contain zipper simplification.*

Proof. The proof goes by induction on the derivations of normalization, path-typing and subtyping (as they are mutually recursive). We show some of the interesting rules here:

Normalization – $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}' \wedge \overset{\omega}{\vdash} \Gamma : \Gamma^\omega \wedge \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \implies \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}.\mathcal{C}$

- Norm-S-Zip – Immediate by induction hypothesis
- Norm-S-Trans-Some – We have:

$$\frac{\Gamma \vdash \mathtt{S} \downarrow (= P' < \mathtt{S}')(\mathbf{1})}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')}\ \textsc{Norm-S-Trans-Some}$$

  By hypothesis, we also have $\Gamma^\omega \overset{\omega}{\vdash} (= P < \mathtt{S}) : (\tau, \mathcal{C})$ (**2**) By inversion, we have:

  $$\Gamma^\omega \overset{\omega}{\vdash} P : (\tau_0, \mathcal{C}_0) \qquad \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1)\ (\mathbf{3}) \qquad \Gamma^\omega \overset{\omega}{\vdash} (\tau_0, \mathcal{C}_0) < (\tau, \mathcal{C})\ (\mathbf{4})$$

  Where $(\tau_1, \mathcal{C}_1)[\overline{\tau} \mapsto \overline{\alpha}] = (\tau, \mathcal{C})$. Using the induction hypothesis with (1) and (3), we get $\Gamma^\omega \overset{\omega}{\vdash} (= P' < \mathtt{S}') : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1)$. However, typing a transparent signature does not introduce type variables. Therefore, by inversion, the set of parameters $\overline{\alpha}$ must be empty, which removes the substitution, and we have:

  $$(\tau_1, \mathcal{C}_1)[\overline{\tau} \mapsto \overline{\alpha}] = (\tau_1, \mathcal{C}_1) = (\tau, \mathcal{C})$$

  This implies $\Gamma^\omega \overset{\omega}{\vdash} (= P' < \mathtt{S}') : (\tau, \mathcal{C})$, which finishes this case.

- Norm-S-Trans-None – We have:

$$\frac{\Gamma \vdash P \triangleright P'(\mathbf{1}) \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'(\mathbf{2})}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')}\ \textsc{Norm-S-Trans-None}$$

  By hypothesis, we also have $\Gamma^\omega \overset{\omega}{\vdash} (= P < \mathtt{S}) : (\tau, \mathcal{C})$ (**3**). By inversion, we have:

  $$\Gamma^\omega \overset{\omega}{\vdash} P : (\tau_0, \mathcal{C}_0)\ (\mathbf{4}) \qquad \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1)\ (\mathbf{5}) \qquad \Gamma^\omega \overset{\omega}{\vdash} (\tau_0, \mathcal{C}_0) < (\tau, \mathcal{C})\ (\mathbf{6})$$

  Where $(\tau_1, \mathcal{C}_1)[\overline{\tau} \mapsto \overline{\alpha}] = (\tau, \mathcal{C})$. Using the induction hypothesis with (2) and (5), we get $\Gamma^\omega \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1)$. By the resolution lemma (Lemma 13), we have:

  $$\Gamma^\omega \overset{\omega}{\vdash} P' : (\tau_0, \mathcal{C}_0') \qquad \Gamma^\omega \overset{\omega}{\vdash} \mathcal{C}_0' < \mathcal{C}_0$$

  We apply the rule M-Typ-Sig-Trans:

$$\frac{\Gamma \overset{\omega}{\vdash} P' : (\tau_0, \mathcal{C}_0') \qquad \Gamma \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1) \qquad \Gamma \overset{\omega}{\vdash} (\tau_0, \mathcal{C}_0') < (\tau, \mathcal{C})\ (\mathbf{7})}{\Gamma \overset{\omega}{\vdash} (= P' < \mathtt{S}') : (\tau, \mathcal{C})}\ \textsc{M-Typ-Sig-Trans}$$

  The only remaining premise is (7), which is a consequence of transitivity of subtyping.

- Norm-S-LocalModType – **Sketch**: *Direct property of the environment*
- Norm-S-PathModType – **Sketch**: *By inversion of* $\Gamma^\omega \overset{\omega}{\vdash} P.T : \lambda\overline{\alpha}.\mathcal{C}$ *(the hypothesis), we get*

  $$\Gamma^\omega \overset{\omega}{\vdash} P : (\_\ , \mathit{sig}\ \overline{\mathcal{D}}\ \mathit{end}) \qquad \mathit{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \in \overline{\mathcal{D}}$$

  Besides, we have $\Gamma \vdash P \triangleright \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}$ *and* `module type` $T = \mathtt{S} \in \overline{\mathtt{D}}$. *We conclude by elaboration of path typing.*

Path-Typing – $\Gamma \vdash P : \mathtt{S} \wedge \overset{\omega}{\vdash} \Gamma : \Gamma^\omega \wedge \Gamma^\omega \overset{\omega}{\vdash} P : (\tau, \mathcal{C}) \implies \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : (\tau, \mathcal{C})$

- Typ-P-Arg and Typ-P-Module – **Sketch**: *Direct property of the environment*
- Typ-P-Norm – immediate application of the normalization lemma (Lemma 11)
- Typ-P-Proj – **Sketch**: *Direct application of the projection rule in* M^ω

Subtyping – $\Gamma \vdash \mathtt{S} \leq \mathtt{S}' \wedge \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \wedge \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}'.\mathcal{C}' \implies \Gamma^\omega \overset{\omega}{\vdash} \lambda\overline{\alpha}.\mathcal{C} < \lambda\overline{\alpha}'.\mathcal{C}'$

- Sub-S-Norm – **Sketch**: *Direct consequence of the normalization lemma Lemma 11.*
- Sub-S-Zipper, Sub-S-FctG and Sub-S-Sig – **Sketch**: *Direct consequence of the corresponding rule in* M^ω*.*
- Sub-S-FctA – **Sketch**: *Almost a direct consequence of the corresponding rule in* M^ω*, just needing to show the elaboration of the extended environment*

$$\overset{\omega}{\vdash} \Gamma \uplus Y : \mathtt{S}_a : \Gamma^\omega, \overline{\alpha}, Y : \mathtt{S}_a$$

□

Soundness

Before finally reaching the soundness result, we have a lemma for signature typing:

Lemma 15 (Elaboration of signature typing)**.** *The elaboration of signatures of ZipML implies the elaboration of signatures of* M^ω*. Given* $\mathtt{S}$*,* Γ *and* Γ^ω *such that* $\Gamma \vdash \mathtt{S}$*, then*

$$\overset{\omega}{\vdash} \Gamma : \Gamma^\omega \implies \exists\overline{\alpha}, \mathcal{C}.\ \Gamma \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$$

Proof. The proof goes by induction on the typing derivation:

- Typ-S-ModType – **Sketch**: *Property of the elaboration of environments*
- Typ-S-Str, Typ-S-GenFct, Typ-S-AppFct – **Sketch**: *Immediate by induction, using the corresponding* M^ω *rule.*
- Typ-S-Ascr – **Sketch**: *Direct use of path typing lemma and subtyping lemma.*

The rules for declarations are immediate. □

Theorem 16: Soundness

Typing of a module expression in ZipML implies typing of the same expression in M^ω:

$$\vdash \mathtt{M} : \mathtt{S} \implies \exists\overline{\alpha}, \mathcal{C}.\ \overset{\omega}{\vdash} \mathtt{M} : \lambda\overline{\alpha}.\mathcal{C} \tag{5.2}$$

Proof. We prove

$$\left.\begin{array}{l} \Gamma \vdash^\Diamond \mathtt{M} : \mathtt{S} \\ \overset{\omega}{\vdash} \Gamma : \Gamma^\omega \end{array}\right\} \implies \exists\overline{\alpha}, \mathcal{C}. \left\{\begin{array}{l} \Gamma^\omega \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \\ \Gamma^\omega \overset{\omega}{\vdash} \mathtt{M} : \exists^\Diamond\overline{\alpha}.\mathcal{C} \\ \text{can be zip-erased} \end{array}\right.$$

by induction on the derivations of the hypothesis.

- Typ-M-Norm – We have:

$$\frac{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}(\mathbf{1}) \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'(\mathbf{2})}{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}'}\ \textsc{Typ-M-Norm}$$

As (1) uses the same typing environment as the conclusion, we can use the induction hypothesis directly, and we get $\overline{\alpha}, \mathcal{C}$ such that

$$\Gamma^{\omega} \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ (\mathbf{3}) \qquad \Gamma^{\omega} \overset{\omega}{\vdash} \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C} \ (\mathbf{4})$$

Applying the elaboration of normalization (Lemma 11) with (2) and (3), we get that the elaboration of $\mathtt{S}'$ is the same: $\Gamma^{\omega} \overset{\omega}{\vdash} \mathtt{S}' : \lambda\overline{\alpha}.\mathcal{C}$. We already have $\Gamma^{\omega} \overset{\omega}{\vdash} \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}$ by (4), which can be zip-erased by hypothesis.

- Typ-M-Ascr – We have:

$$\frac{\Gamma \vdash \tilde{\mathtt{S}} \ (\mathbf{1}) \qquad \Gamma \vdash \tilde{P} : \mathbf{S} \ (\mathbf{2}) \qquad \Gamma \vdash \mathbf{S} \leqslant \tilde{\mathtt{S}} \ (\mathbf{3})}{\Gamma \vdash^{\diamond} (\tilde{P} : \tilde{\mathtt{S}}) : \tilde{\mathtt{S}}}\ \textsc{Typ-M-Ascr}$$

We need to find $\overline{\alpha}$ and $\mathcal{C}_1$ such that:

$$\Gamma^{\omega} \overset{\omega}{\vdash} \tilde{\mathtt{S}} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1) \ (\mathbf{4}) \qquad \Gamma^{\omega} \overset{\omega}{\vdash} (= \tilde{P} < \tilde{\mathtt{S}}) : (\tau_1, \mathcal{C}_1) \ (\mathbf{5})$$

By applying elaboration of signature typing Lemma 15 on (1), elaboration of path-typing Lemma 12 on (2) we get:

$$\Gamma^{\omega} \overset{\omega}{\vdash} \tilde{\mathtt{S}} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1) \ (\mathbf{6}) \qquad \Gamma^{\omega} \overset{\omega}{\vdash} \tilde{P} : (\tau, \mathcal{C}) \ (\mathbf{7}) \qquad \Gamma^{\omega} \overset{\omega}{\vdash} \mathbf{S} : (\tau, \mathcal{C})$$

(6) gives us the first goal (4). We apply the elaboration of subtyping Lemma 14 and get $\Gamma^{\omega} \overset{\omega}{\vdash} (\tau, \mathcal{C}) < \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1)$, which, by inversion, gives us a list of type expressions $\overline{\tau}$ such that:

$$\Gamma^{\omega} \overset{\omega}{\vdash} (\tau, \mathcal{C}) < (\tau_1, \mathcal{C}_1)[\overline{\alpha} \mapsto \overline{\tau}] \ (\mathbf{8})$$

We use M-Typ-Mod-Ascr to show (5) (the hypothesis being exactly (7), (6) and (8):

$$\frac{\Gamma \overset{\omega}{\vdash} P : (\tau, \mathcal{C}) \qquad \Gamma \overset{\omega}{\vdash} \tilde{\mathtt{S}} : \lambda\overline{\alpha}.(\tau_1, \mathcal{C}_1) \qquad \Gamma \overset{\omega}{\vdash} (\tau, \mathcal{C}) < (\tau_1, \mathcal{C}_1)[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\omega}{\vdash} (P : \tilde{\mathtt{S}}) : \exists^{\nabla}\overline{\alpha}.(\tau_1, \mathcal{C}_1)}\ \textsc{M-Typ-Mod-Ascr}$$

- Typ-M-Path **Sketch**: *Just a use of the path typing lemma Lemma 12*
- Typ-M-FctG, Typ-M-AppG, Typ-M-Str, Typ-M-Mode: Immediate by induction hypothesis, as there is a directly corresponding M^{ω} rule.
- Typ-M-FctA **Sketch**: *We have:*

$$\frac{\Gamma \vdash \mathtt{S}_a \ (\boldsymbol{1}) \qquad \Gamma \uplus Y : \mathtt{S}_a \vdash^{\nabla} \mathtt{M} : \mathtt{S} \ (\boldsymbol{2})}{\Gamma \vdash^{\diamond} (Y : \mathtt{S}_a) \to \mathtt{M} : (Y : \mathtt{S}'_a) \to \mathtt{S}}\ \textit{Typ-M-FctA}$$

Induction hypothesis directly on (1) which gives: $\Gamma^{\omega} \overset{\omega}{\vdash} \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a$ (*3*). *Before using the induction hypothesis on (2), we need to show*

$$\overset{\omega}{\vdash} (\Gamma \uplus Y : \mathtt{S}_a) : (\Gamma^{\omega}, \overline{\alpha}, Y : \mathcal{C}_a)$$

which is easy by environment elaboration and using (3). We get:

$$\Gamma^\omega, \overline{\alpha}, Y : \mathcal{C}_a \overset{\omega}{\vdash} \mathtt{M} : \exists^\nabla \overline{\beta}.\mathcal{C} \quad (4) \qquad \Gamma^\omega, \overline{\alpha}, Y : \mathcal{C}_a \overset{\omega}{\vdash} \mathtt{S} : \lambda\overline{\beta}.\mathcal{C} \quad (5)$$

Let us set $\mathcal{C}' \triangleq \mathcal{C}\left[\overline{\beta \mapsto \beta'(\overline{\alpha})}\right]$ *We need to show:*

$$\Gamma^\omega \overset{\omega}{\vdash} (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^\nabla \overline{\beta'}.\, \beta'(\overline{\alpha}) \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}' \quad (6) \qquad \Gamma^\omega \overset{\omega}{\vdash} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}' \quad (7)$$

For (6), we use M-Typ-Mod-AppFct, which hypothesis are exactly (3) and (5). For (7), we use M-Typ-Sig-AppFct, which hypothesis are exactly (3) and (5).

□

5.5.2 Zipper renaming, introduction, and extrusion

These operations are not directly available in ZipML. Yet, we could add them as additional typing rules.

Renaming of zipper labels

When a zipper is introduced during projection, it turns a self-reference into a label whose name is chosen arbitrarily and cannot later be changed. Hence, it must be chosen fresh so as to avoid shadowing of other labels of the same name, which well-formedness forbids. Still, labels can be renamed *consistently*. For example, if $\Gamma \vdash \mathtt{M} : \langle A : \mathtt{D} \rangle\, \mathtt{S}$ and A does not appear free in Γ, then we also have

$$\Gamma \vdash \mathtt{M} : \langle A' : \mathtt{D}[A \mapsto A'] \rangle\, \mathtt{S}[A \mapsto A']$$

when A' is chosen fresh for Γ, $\mathtt{D}$, and $\mathtt{S}$. This is a lemma and not a *derivable* typing rule. However, we can then safely add renaming as an *admissible* typing rule.

A logically cleaner solution, worth considering in the future, would be to introduce a binder construct $\nu A.\, \mathtt{S}$ in signatures that limits the scope of A to $\mathtt{S}$. However, this would be more involved and invasive as 1. typing rules should then be adjusted to extrude binders and gather them at to the toplevel of typing judgments and in the codomains of generative functors, and 2. equivalence (an other judgments) should also be redefined up to the consistent renaming of binders.

Introduction of zippers

$$\textsc{Typ-M-ZipperI}\quad \frac{\Gamma \vdash \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \langle A : \varnothing \mid \overline{\mathtt{D}} \rangle\, \mathtt{S} : \mathsf{wf}}{\Gamma \vdash \mathtt{M} : \langle A : \varnothing \mid \overline{\mathtt{D}} \rangle\, \mathtt{S}}$$

In ZipML, zippers are only introduced during projections and indirectly transported by other operations. They can be simplified at toplevel. Yet, they cannot be moved, in particular, they cannot be extruded. In ZipML^F, we may allow the introduction of a right-zipper as a toplevel typing rule. The combination of both premises ensure that the zipper label does not appear in Γ nor anywhere in $\mathtt{S}$. This is always safe, as the extra fields $\overline{\mathtt{D}}$, are not currently accessible from $\mathtt{S}$ and, intuitively, will not be accessible by the rest of the program.

Interesting, the rule may then be followed by an equivalence that moves the right floating fields on the left and do some "rewiring" in $\mathtt{S}$, accordingly. For example, we could turn a module of signature $\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}$ into one of type $\langle A : \overline{\mathtt{D}} \rangle\, \mathtt{sig}\ \overline{\mathtt{D}}\ /\!/\ A\ \mathtt{end}$ by first introducing $\langle A : \varnothing \mid \overline{\mathtt{D}} \rangle\, \mathtt{S}$, then using equivalence and simplification. This could also be used for zipper extrusion.

For example, if $\Gamma \vdash \mathtt{M} : \mathtt{S}_0$ where $\mathtt{S}_0$ is $\mathtt{sig}_A\ \mathtt{module}\ X : \langle A_0 : \mathtt{D} \rangle\, \mathtt{S}\ \mathtt{end}$ and A does not appear in $\mathtt{D}$, then we could extrude $A_0 : \mathtt{D}$ by giving $\mathtt{M}$ the successive types:

- $\mathtt{S}_0 \triangleq \mathtt{sig}_A\ \mathtt{module}\ X : \langle A_0 : \mathtt{D}\rangle\ \mathtt{S}\ \mathtt{end}$ *initially*
- $\langle A_1 : \varnothing \mid \mathtt{D} \mathbin{//} A_1.\mathbb{Z}.X.A_0\rangle\ \mathtt{sig}_A\ \mathtt{module}\ X : \langle A_0 : \mathtt{D}\rangle\ \mathtt{S}\ \mathtt{end}$ *by introduction of a zipper*
- $\langle A_1 : \mathtt{D}\rangle\ \mathtt{sig}_A\ \mathtt{module}\ X : \langle A_0 : \mathtt{D} \mathbin{//} A_1\rangle\ \mathtt{S}\ \mathtt{end}$ *by equivalence*
- $\langle A_1 : \mathtt{D}\rangle\ \mathtt{sig}_A\ \mathtt{module}\ X : \langle A_0 : \mathtt{D} \mathbin{//} A_1\rangle\ \mathtt{S}\,[A.X.A_0 \leftarrow A_1]\ \mathtt{end}$ *by equivalence*
- $\langle A_1 : \mathtt{D}\rangle\ \mathtt{sig}\ \ \mathtt{module}\ X : \langle A_0 : \varnothing \mid \mathtt{D} \mathbin{//} A_1\rangle\ \mathtt{S}\,[A.X.A_0 \leftarrow A_1]\ \mathtt{end}$ *by equivalence*
- $\langle A_1 : \mathtt{D}\rangle\ \mathtt{sig}\ \ \mathtt{module}\ X : \mathtt{S}\,[A.X.A_0 \leftarrow A_1]\ \mathtt{end}$ *by projection*
- $\mathtt{S}_1 \triangleq \langle A_0 : \mathtt{D}\rangle\ \mathtt{sig}\ \mathtt{module}\ X : \mathtt{S}\,[A.X.A_0 \leftarrow A_0]\ \mathtt{end}$ *by renaming of* A_1.

5.5.3 Completeness of ZipML with respect to M^ω

Conversely, one may wonder whether the type system of ZipML is powerful enough to encompass M^ω. We argue that it is indeed the case. In this section we present some key properties supporting this claim. Namely, we remark that:

1. floating fields can be used to encode existentially quantified variables
2. narrow subtyping on zippers can simulate the M^ω extrusion and skolemization of variables
3. we do not need to encode universally and lambda bound variables

Encoding existentially quantified types with floating fields

Our claim is that *floating fields can encode all existentially quantified variables of* M^ω. We first consider first-order type variables, then module identities, and finally higher-order types.

First-order type variables Let us consider a M^ω-signature with a single quantified type variable of the base kind $\star$, which is of the form $\exists\alpha.\mathcal{C}$. Inside $\mathcal{C}$, the variable α is accessible everywhere. This is quite similar to a zipper signature $\langle A : \mathtt{type}\ t = A.t\rangle\ \mathtt{S}$, where $A.t$ is a type accessible everywhere inside $\mathtt{S}$. Therefore, we could translate the M^ω-signature into a zipper signature by introducing a floating field with a fresh name for every quantified variable. We may define a reverse elaboration judgment $\Gamma^\omega \vdash \exists\overline{\alpha}.\mathcal{C} \hookleftarrow \langle\gamma\rangle\ \mathtt{S}$, where we extend the environment to attach the name of a floating field to every (existentially) quantified abstract type. We would have rules of the form:

$$\textsc{Rev-S-Star}\ \frac{\Gamma, \alpha \hookleftarrow A.t \vdash \mathcal{C} \hookleftarrow \mathtt{S} \qquad A\ \mathsf{fresh}}{\Gamma \vdash \exists\alpha.\mathcal{C} \hookleftarrow \langle A : \mathtt{type}\ t = A.t\rangle\ \mathtt{S}} \qquad \textsc{Rev-T-AbsType}\ \frac{\alpha \hookleftarrow A.t \in \Gamma^\omega}{\Gamma \vdash \alpha \hookleftarrow A.t}$$

Module identities Modules identities of M^ω can be encoded as floating module fields. However, there is a difficulty: what is the attached signature of an identity floating field? The simplest answer is to extend ZipML with a *bottom signature* $\bot$ that is a subtype of all signatures. Doing so, we would get rules of the following form:

$$\textsc{Rev-S-Mod}\ \frac{\Gamma, \alpha_{id} \hookleftarrow A.X \vdash \mathcal{C} \hookleftarrow \mathtt{S} \qquad A\ \mathsf{fresh}}{\Gamma \vdash \exists\alpha_{id}.\mathcal{C} \hookleftarrow \langle A : \mathtt{module}\ X : \bot\rangle\ \mathtt{S}} \qquad \textsc{Rev-S-Transparent}\ \frac{\alpha_{id} \hookleftarrow A.X \in \Gamma^\omega \qquad \Gamma \vdash \mathcal{C} \hookleftarrow \mathtt{S}}{\Gamma \vdash (\alpha_{id}, \mathcal{C}) \hookleftarrow (= A.X < \mathtt{S})}$$

However, as hinted in Section 4.2.3, module identities of M^ω are always attached to signatures that have a common ancestor in the subtyping order. Therefore, rather than extending ZipML with a bottom signature $\bot$, we could (lightly) instrument the typing rules of M^ω to obtain the common ancestor signature associated with each module identity and use it in the floating field that encodes this identity.

Higher-order types Higher-order types of M^ω can be encoded as floating functors producing a single type field. Again, there is a subtlety: what is the signature of the domain of such functor? The simplest answer is to extend ZipML with a *top signature* $\top$ that is a supertype of all signatures. Using it, we would get rules of the form:

$$\textsc{Rev-S-HigherOrder}\quad \frac{\Gamma, \varphi \hookleftarrow A.F(\cdot).t \vdash \mathcal{C} \hookleftarrow \mathtt{S}}{\Gamma \vdash \exists^{\blacktriangledown} \varphi\,.\mathcal{C} \hookleftarrow \langle A : \mathtt{module}\ F : (Y : \top) \to \mathtt{sig}_B\ \mathtt{type}\ t = B.t\ \mathtt{end}\rangle\ \mathtt{S}}$$

$$\textsc{Rev-T-Transparent}\quad \frac{\varphi \hookleftarrow A.F(\cdot).t \in \Gamma^\omega \qquad \alpha_{id} \hookleftarrow B.X}{\Gamma \vdash \varphi(\alpha_{id}) \hookleftarrow A.F(B.X).t}$$

Higher-order module identities would work similarly. Following the same reasoning as for module identities, we conjecture that, rather than extending ZipML with a top signature $\top$, we could instrument the typing rules of M^ω to obtain the domain of the functor that originally introduced φ, which is a supertype of all use-cases.

Universally and lambda quantified types Our argument applies to existentially quantified types. But the universal quantification and lambda quantification of M^ω are always used for signatures that come from elaboration of the source, and can therefore also be represented in ZipML, without using floating field to encode type variables.

Extrusion

Floating fields can be extruded, similarly to existential types. For instance, if floating fields have been introduced in front of submodules, we could extrude them:

$$\begin{array}{rl} \text{if} & \mathtt{M} : \mathtt{sig}_A\ \mathtt{module}\ X_1 : \langle A_1 : \mathtt{type}\ t = A_1.t\rangle\ \mathtt{sig}\ \mathtt{type}\ t = A_1.t\ \mathtt{end} \\ & \qquad \mathtt{module}\ X_2 : \langle A_2 : \mathtt{type}\ t = A_2.t\rangle\ \mathtt{sig}\ \mathtt{type}\ t = A_2.t\ \mathtt{end}\ \mathtt{end}[A] \\ \text{then} & \mathtt{M} : \langle A_1 : \mathtt{type}\ t = A_1.t\,;\, A_2 : \mathtt{type}\ t = A_2.t\rangle \\ & \qquad \mathtt{sig}_A\ \mathtt{module}\ X_1 : \mathtt{sig}\ \mathtt{type}\ t = A_1.t\ \mathtt{end}\ \mathtt{module}\ X_2 : \mathtt{sig}\ \mathtt{type}\ t = A_2.t\ \mathtt{end}\ \mathtt{end}[A] \end{array}$$

This also applies to *skolemization*, as we could also have:

$$\begin{array}{rl} \text{if} & \mathtt{M} : (Y : \mathtt{S}_a) \to \langle B : \mathtt{type}\ t = B.t\rangle\ \mathtt{sig}\ \mathtt{type}\ t = B.t\ \mathtt{end} \\ \text{then} & \mathtt{M} : \langle B : \mathtt{module}\ F : (Y_0 : \mathtt{S}_a) \to \mathtt{sig}_D\ \mathtt{type}\ t = D.t\ \mathtt{end}\rangle\ (Y : \mathtt{S}_a) \to \mathtt{sig}\ \mathtt{type}\ t = B.F(Y).t\ \mathtt{end} \end{array}$$

where we would first introduce the outer zipper as a right floating field, apply an equivalence, and remove the inner zipper by projection. (Here, for the domain of the floating functor, we could also use the top signature $\top$ instead of the signature $\mathtt{S}_a$ of the functor we extruded from, as we did.)

Overall, equivalence over floating fields allows us to mimic the extrusion and skolemization mechanisms of M^ω. Informally, during a proof a completeness, it would allow us to maintain a compositional correspondence between M^ω and ZipML every time there is extrusion or skolemization happening. At each step, we would be able to combine the ZipML signatures obtained by induction hypothesis and use equivalence to make them correspond to M^ω.

Comparing reverse translation and anchoring Technically, the reverse translation drafted here could be composed with zipper simplification. We conjecture that it would produce the same result as using M^ω-anchoring when it succeeds, up to normalization.

5.5.4 Other properties

Normalization is a floating typing rule that can be called anytime. Normalization itself may be performed by need, but also in a strict manner. It is therefore left to the implementation to normalize just as necessary – as one would typically do with β-reduction.

As a result, the inferred signature is not unique, returning different syntactic answers according to the amount of normalization that has been done. Hence, we may have $\Gamma \vdash \mathtt{M} : \mathtt{S}$ and $\Gamma \vdash \mathtt{M} : \mathtt{S}'$ when $\mathtt{S}$ and $\mathtt{S}'$ syntactically differ – even a lot! as one may contain a signature definition expanded in the other.

Still, we should then have $\Gamma \vdash \mathtt{S}' \approx \mathtt{S}''$. That is, the inferred signatures should only differ up to their presentation, but remain inter-convertible – and otherwise simplified in the same manner.

One might expect a stronger result, stating that there is a best presentation where module names would have been expanded as little as possible. This would be worth formalizing, although a bit delicate. In particular, we probably wish to keep names introduced by the user, but not let the algorithm reintroduce a name when it recognized an inferred signature that was not named, but just happens to be equivalent to one with a name.

5.6 Discussion

Zipper renaming

When a zipper is introduced during projection, it turns a self-reference into a label whose name is chosen arbitrarily and cannot later be changed. Hence, it must be chosen fresh so as to avoid shadowing of other labels of the same name, which well-formedness forbids. Still, labels can be renamed *consistently*. For example, if $\Gamma \vdash \mathtt{M} : \langle A : \mathtt{D} \rangle\, \mathtt{S}$ and A does not appear free in Γ, then we also have

$$\Gamma \vdash \mathtt{M} : \langle A' : \mathtt{D}[A \mapsto A'] \rangle\, \mathtt{S}[A \mapsto A']$$

when A' is chosen fresh for Γ, $\mathtt{D}$, and $\mathtt{S}$. This is a lemma and not a *derivable* typing rule. However, we can then safely add renaming as an *admissible* typing rule.

Renaming could be internalized in two ways. A light solution is to add an explicit code-free construct [11] $[A \leftarrow A']\,\mathtt{M}$ with the obvious typing rule:

$$\textsc{Renaming} \quad \frac{\Gamma \vdash \mathtt{M} : \langle A : \mathtt{D} \rangle\, \mathtt{S} \qquad A' \notin \Gamma, \mathtt{D}, \mathtt{S}}{\Gamma \vdash \mathtt{M}\,[A \leftarrow A'] : \langle A' : \mathtt{D}[A \mapsto A'] \rangle\, \mathtt{S}[A \mapsto A']}$$

Renaming could break further typing, and it would be the user's responsibility to use renaming appropriately, but it would preserve soundness, thanks to the meta-theoretical renaming property.

A logically cleaner solution, worth considering in the future, would be to introduce a binder construct $\nu A.\, \mathtt{S}$ in signatures that limits the scope of A to $\mathtt{S}$. However, this would be more involved and invasive as 1. typing rules should then be adjusted to extrude binders and gather them at to the toplevel of typing judgments and in the codomains of generative functors, and 2. equivalence (an other judgments) should also be redefined up to the consistent renaming of binders.

Full zippers

As full zippers play an important role in the proof of preservation of typing in Section 5.4.8, one can wonder if it would be relevant to work in ZipML^F directly, instead of ZipML. So far,

[11] That is, $[A \leftarrow A']\,\mathtt{M}$ behaves as $\mathtt{M}$ at runtime.

we consider that ZipML^F is a useful tool for the proof, as the reorganization of zippers can be seen as an equivalence. It would be worth exploring if seeing ZipML as the projection of ZipML^F would have other benefits.

Chapter 6

Advanced features

In this chapter, we step back from M^{ω} and ZipML to discuss other features of ML modules that were not formally treated in this work. This chapter can be seen as the continuation of Chapter 2, and we reuse the feature markers (C, F, P, ✔, ✔, ✚).

Overview In Section 6.1, we discuss the *composition* between modules, i.e., the way modules can be combined. After presenting hierarchical and flat composition, we briefly discuss recursive modules and open recursion, both from a typing and semantic point of view.

In Section 6.2, we present other advanced features, starting with those enabling a better interaction between the module language and the core language, namely *first-class modules* and *modular explicits, modular implicits*. Then, we move on to the features for signature manipulation and miscellaneous ones.

6.1 Composition

Composition refers to the ways different modules can be combined. In this section we propose a *taxonomy* to classify the forms of composition and briefly discuss the associated features.

Forms of composition Let us assume that we want to compose two modules, $\texttt{M}_1$ and $\texttt{M}_2$. We propose the following criteria:

1. **Delayed vs Immediate**: If we know the modules that are composed, here $\texttt{M}_1$ and $\texttt{M}_2$, the composition is *immediate*. If instead, we want to compose $\texttt{M}_2$ with a module that is not chosen yet, we call the composition *delayed*. In delayed composition, the user must provide the signature of what is expected of the other module. Delayed composition is associated with a *resolution* where the module $\texttt{M}_1$ is actually chosen[1]. This resolution can be made at another point than the definition of $\texttt{M}_2$, and it allows $\texttt{M}_1$ and $\texttt{M}_2$ to be compiled separately. It also makes it possible to create several instances by composing $\texttt{M}_2$ with different modules. Overall, delayed composition has some flexibility but requires a signature annotation, and therefore, forces a *fixed view* of the other module.

2. **Sequential vs Recursive**: If $\texttt{M}_2$ depends (either for value definitions or type definitions) on $\texttt{M}_1$ but not the other way around, we can compose them in a *sequence*: first $\texttt{M}_1$ then $\texttt{M}_2$. If the dependency is mutual, i.e., $\texttt{M}_1$ depends on $\texttt{M}_2$ and $\texttt{M}_2$ depends on $\texttt{M}_1$, we need a *recursive* composition. Expressivity-wise, sequential composition is a sub-case of recursive composition, which is more general. However, with recursive composition comes the problem of well-foundation: how can we ensure that both type definitions and value definitions are not cyclic (in a ill-formed way)? We explore this challenge (and other practical ones) in Section 6.1.2. Overall, this added complexity explains why the simplicity of sequential composition is still appealing for users and language designers.

3. **Hierarchical vs Flat**: When $\texttt{M}_1$ and $\texttt{M}_2$ are structures (not functors), another distinction arises. After the composition of $\texttt{M}_1$ and $\texttt{M}_2$, if the content of both are in separate namespaces, the composition is called *hierarchical*: the namespace hierarchy keeps a trace of the composition. If instead, the contents of the two modules are merged in the same namespace, the composition is called *flat*. Flat composition can create clashes of namespaces, i.e., *shadowing*. Overall, flat composition is more general than hierarchical, as the desired name-space hierarchy can always be explicitly added before composition, but at the cost of dealing with shadowing. We discuss flat composition in more details in Section 6.1.1.

We sum up the associated features in Figure 33.

6.1.1 Hierarchical and flat composition

Hierarchical

The easiest form of composition is *closed-sequential-hierarchical* composition, which is provided by module bindings. The user can combine modules, but they remain in different name-spaces, with different prefixes, as illustrated by the following code:

```
1 module X1 = struct let x = 42 end
2 module X2 = struct let x = 43 end
3 module Z = struct module X = X1 module X' = X2 end
```

[1] At this point, a subtyping check is made between what was expected of $\texttt{M}_1$ and what is actually provided. $\texttt{M}_1$ is then seen through either an opaque or transparent ascription, depending on the features. For functor calls, the effective ascription is transparent.

		Sequential		Recursive
		Hierarchical	Flat	
Immediate		submodule	include/open	recursive modules
Delayed	creation	functor		mixin modules[b]
	resolution	application	include functor[a]	

Figure 33: Summary of the language constructions for composition.
[a]Experimental feature implemented in an extension of OCaml
[b]Implemented in MixML [Rossberg and Dreyer, 2013], an experimental language based on mixin composition

```
module X1 = struct let x = 42 end
module X2 = struct let x = 43 end
module Z : sig
  module X : sig val x : int end
  module X' : sig val x : int end
end
```

The module `Z` is the result of the composition of `X1` and `X2`. The content of `X1` are in a hierarchically different namespace from the content of `X2`, with a different prefix: `X1.x` vs `X2.x`.

Open-hierarchical-composition is provided by functors. The *creation* is definition of the functor, that depends on a *yet-unresolved* module parameter. At the point of functor application, the module is resolved, as illustrated by the following code:

```
(* Creation *)
module F (Y : sig val x : int end) = struct let z = Y.x + 42 end
module X  = struct let x = 1 end
module X' = struct let x = 2 let y = 3 end
(* Resolution *)
module Z = F(X)
module Z' = F(X')
```

```
module F : functor (Y : sig val x : int end) → sig val z : int end
module X : sig val x : int end
module X' : sig val x : int val y : int end

module Z : sig val z : int end
module Z' : sig val z : int end
```

The creation of `F`, `X` and `X'` can be in different files and in any order. The functor can be used several times, creating different instances `Z` and `Z'`. Inside the body of the functor, the content of the parameter is hierarchically separated (prefixed by `Y`). It is not re-exported by default.

Flat composition

There are two main language constructs for *closed-flat-composition*, that we detail below.

Include The first construct is **`include`**. It takes all the fields of a structure and put them, *at the same level*, inside the current structure. It can be used for modules expressions:

```
module X1 = struct let x1 = 42 end
module X2 = struct let x2 = 43 end
module Z = struct include X1 include X2 let z = x1 + x2 end
```

```
module X1 = sig val x1 : int end
module X2 = sig val x2 : int end
module Z : sig val x1 : int val x2 : int val x : int end
```

It can be used with anonymous modules, as long as they have a structural signature. It is also available for combining signatures. As flat composition merges two name-spaces, there can be shadowing between the two. In OCaml, shadowing *via* an include statement is disallowed and produces a typechecking error. A typical use-case for **include** is to *extend* a module with additional fields.

Open The second construct for flat composition is a variant of the **include** statement where the imported bindings are made available in the current structure but not re-exported: **open**.

```
module X = struct let x = 42 end
module Y = struct open X let y = x + 1 end
```

```
module X : sig val x : int end
module Y : sig val y : int end (* no field x *)
```

While the restricted **open** `P` statement (restricted to paths) is straightforward, the generalized statement **open** `M` [Li and Yallop, 2017] (where `M` is any module expression) is more subtle: it may create signature avoidance situations by opening abstract type declarations that are not exported and yet appear in the signature.. For instance, the following program fails to typecheck:

```
module X = struct
  open (struct type t = A | B end)
  let x : t = A
end
```

```
Error: The type t/1676 introduced by this open appears in the signature
       Line 3, characters 6-7:
         The value x has no valid type if t/1676 is hidden
```

As for **include**, opening a module can lead to shadowing of fields. While it is again disallowed by default in OCaml, the language also offers a variant **open!** that allows shadowing.

Open-flat composition

There is a missing construct for open-flat composition. The closest we can do in OCaml would be to combine a functor with an include statement, as in:

```
module type S = sig val x : int end
module F = functor (Y:S) → struct include Y let f = x + x end
module X = struct let x = 42 let y = 41 end
module Z = F(X)
```

```
module Z : sig val x : int (* from X *) val f : int (* from F *) end
```

Technically, it is an open flat composition, as the content of `X` and `F` are indeed merged in the same namespace at the resolution point. However, this is not satisfactory, as `X` is restricted to the *fixed view* `S`: `F` can be used only to extend it beyond the field of `S`, other extra-fields are lost (here, the field `y` of `X` is not present in `Z`). Due to considerations of compilation-schemes (discussed in Section 2.2.4), this implicit (transparent) ascription at functor call could not be removed. Alternatively, we can use a temporary structure and flat-compose the argument together with the result of functor application, as in:

```
module Z = struct
  open (struct module Temp = struct let x = 42 let y = 41 end end)
  include Temp
  include F(Temp)
end
```

```
module Z : sig val x : int val y : int (* from Temp *) val f : int (* from F *) end
```

Again, this pattern is not fully satisfactory, as it (1) requires to use a dummy name `Temp` and (2) does not compose well: every include of a functor call would require a new temporary structure, with the verbose pattern of line 3.

Include functor A proposal[2] has been made for OCaml (and is already implemented in JaneStreet's fork of OCaml) for a new feature called **`include functor`**, aimed at solving this problem. It provides a way to do the *resolution* of open-flat composition, combining a structure and the result of a functor call:

```
1 module Z = struct
2   let x = 42
3   let y = 41
4   include functor F (* = include F(current structure up to this point) *)
5 end
```

```
1 module Z : sig val x : int val y : int val f : int end
```

Using the combined keywords **`include functor`** it allows to flat-compose the result of applying `F` to the current structure being built.

6.1.2 Recursive composition

Sequential composition does not allow the user to define modules that have mutual dependencies. This is a strong limitation of modularity: while the core-language provides mutually recursive types and values, their definitions cannot cross module boundaries. As stated by Russo [2001]: "This limitation compromises modular programming, forcing the programmer to merge conceptually (i.e. architecturally) distinct modules." To overcome this limitation, several proposal were made, most notably *recursive modules* (for immediate composition) and *mixin modules* (for delayed composition) Despite a strong academic interest [Rossberg and Dreyer, 2013; Im et al., 2011; Montagu and Rémy, 2009; Dreyer, 2007a,b; Nakata and Garrigue, 2006; Russo, 2001; Crary et al., 1999; Jaakkola, 2020; Montagu and Rémy, 2009] (to name a few), recursive modules were implemented only in OCaml Leroy [2003] and Moscow ML [Russo, 2001], while mixins were implemented in MixML [Rossberg and Dreyer, 2013]. As is was not at the heart of this thesis, we only briefly present existing features, challenges, or research ideas for future work. We start with an example.

Recursive modules in OCaml OCaml offers a notion of *recursive module binding*, introduced by the keywords **`module rec`**. Each recursive module must be given an explicit signature. The pattern is given on the left-hand side, with the result of the typechecking, a *recursive module declaration*, being written on the right-hand side:

```
1 module rec X1 : S1 = M1
2        and X2 : S2 = M2
3        ...
4        and Xn : Sn = Mn
```

```
1 module rec X1 : S1
2        and X2 : S2
3        ...
4        and Xn : Sn
```

A typical use-case for recursive modules is when a type definition relies on another type definition that is provided by a functor call on the current structure. An example, adapted from [Nakata and Garrigue, 2006], is the following:

```
1 module rec Tree :
2   sig type t = L | N of TreeSet.t val compare : t → t → int end =
3   struct
4     (* Leafs or sets of sub-trees *)
5     type t = L | N of TreeSet.t
6     let compare (t1: t) (t2 : t) = match (t1, t2) with
```

[2] See RFC at https://github.com/ccasin/RFCs/blob/include-functor/rfcs/include_functor.md

```
7       | (Leaf, Leaf) → 0
8       | (Node(_), Leaf) → 1
9       | (Leaf, Node(_)) → -1
10      | (Node(s1), Node(s2)) → TreeSet.compare s1 s2
11    end
12     and TreeSet : (Set.S with type elt = Tree.t) = Set.Make(Tree)
```

```
1 module rec Tree : sig type t = L | N of TreeSet.t val compare : t → t → int end
2     and TreeSet : sig type t type elt = Tree.t val compare : t → t → int ... end
```

Here, the definition of `Tree.t` at line (3) relies on sets of itself at each node `TreeSet.t`, where sets are provided by the `Set` functor. Both the type definitions and value bindings are mutually recursive between `Tree` and `TreeSet`.

Challenges of recursive composition We identify four main challenges that are inherent to recursive composition and that we detail in the rest of this section.

Double-Vision

An issue that appears with the interaction of type abstraction and recursive definitions ([Dreyer, 2007a,b; Crary et al., 1999; Rossberg and Dreyer, 2013]) is the *double vision* problem. Let us see a example in OCaml:

```
1 module rec X : sig type t val x : t end
2   = struct type t = int let x = (Y.f 0) + 1 end
3 and Y : sig val f : X.t → X.t end
4   = struct let f x = x end
```

```
1 Error: This expression has type t but an expression was expected of type X.t
```

The outside vision of `t` is abstract, while the vision inside the structure is `type t = int`. As OCaml does not have double vision, it does not identify `X.t` and `t` and fails to typecheck. The problem has been solved by Dreyer [2007b] (and later extended to *crossed-eyes double vision* [Rossberg and Dreyer, 2013]), via a two-pass algorithm that introduces abstract type variables and refined them to their definition only inside the structure that defines them.

Cyclic type definitions

As type definitions inside recursive modules can be mutually recursive, it is possible to define type cycles. This example produces a stack overflow in current OCaml:

```
1 module rec X : sig type t = Y.t end = struct type t = int end
2        and Y : sig type t = X.t end = struct type t = int end
```

Different meta-theories (mainly equi-recursive or iso-recursive types), allow to handle different forms of types cycles.

Recursive signature wellformedness

Even in the absence of type cycles, checking mutually recursive definitions is non trivial. Let us assume that we want to check `S1` and `S2` that are mutually recursive. Checking `S1` requires information from `S2`, but we cannot trust `S2` yet and put it in the typing environment. And vice versa for checking `S2`. In the absence of applicative functors, a trick can be used: signatures can be first simplified by making all type fields abstract. These approximate signatures `S1'` and `S2'` can be easily checked for wellformedness and put in the typing environment. With applicative functors however, there is a need for a more involved solution. In OCaml, this code fails to typecheck:

```
module rec X : sig type t = int end = struct type t = int end
and Y : sig type u = F(X).t end = struct type u = F(X).t end;;
```

Safe initialization of recursive structures

A module under initialization does not fulfill its signature specification yet and can be unsafe to use, as it may have uninitialized fields. Yet, during initialization of mutually recursive modules, accesses to other modules under initialization can be made. Ensuring that such accesses are well-ordered and well-founded is a known challenge, both in the ML community ([Hirschowitz and Leroy, 2005; Reynaud et al., 2021]), Haskell and in the Object Oriented Programming community ([Boudol, 2004; Liu et al., 2021, 2020; Qi and Myers, 2009; Blaudeau and Liu, 2022; Syme, 2006; Liu, 2020; Summers and Mueller, 2011; Liu et al., 2023]). In ML modules, most systems have a *back-patched* semantics: fields are first filled with a dummy value that is then patched to the right one when it is actually computed. In OCaml, functions are first initialized with dummy function that throws an exception `Undefined_recursive_module`. This exception can be raised during evaluation:

```
module rec X : sig val x : unit → int end =
  struct let x () = Y.y + 1 end
and Y : sig val y : int end =
  struct let y = X.x () + 1 end
```

```
Fatal error: exception Undefined_recursive_module
```

Or it can be raised later, if the dummy function is never back-patched:

```
module rec X : sig val f : unit → int end = struct include X end
(* later *)
let fail = X.f ()
```

6.2 Other features

In this section, we briefly present the remaining features that have not been covered yet. We start with the interaction between the core and module language in Section 6.2.1. In Section 6.2.2, we discuss the features for signature manipulation.

6.2.1 Core and module language interactions

The separation between the core and module languages makes modules *second-class* citizens: they cannot be handled directly by core-language expressions. We present some features designed to facilitate interactions between the two language layers.

Local modules The stratification prevents interleaving module and value definitions. To allow for the definition of a module within a value definition, OCaml features a *local module* construct, using the **`let module`** keyword:

```
let f (x:int) =
  let module M = struct let y = 42 + x end in
  M.y + x
```

```
val f : int → int = <fun>
```

Similarly to the **`open`** construct (see Section 6.1.2), the resulting type cannot refer to the locally bound module, as it would otherwise escape it scope. For instance:

```
let f (x: int) =
  let module M = struct open struct (type t = A of int | B of bool end) in
```

```
  M.A(x + 1)
Error: The type constructor M.t would escape its scope
```

This constitutes a limited case of *avoidance*, as we only need to avoid names in a core-language type, not a whole signature[3]. Local modules become useful if the core-language can handle module directly, which is the goal of the next feature.

First-class modules So far, modules are considered as *second-class citizens* of the core-language: they cannot be stored as values nor returned by functions. This makes the module language completely separated from the core language: there is no way to choose a module based on a dynamic value. To remedy this problem, *first-class* modules[4] were introduced by Russo [2000] and well studied since then [Dreyer et al., 2003; Rossberg et al., 2014; Rossberg, 2006] (among other). They were added in OCaml 3.12. There are two key constructs: one to turn a module into a core-language value (making it *first-class*) and one to turn a core-language value back into a module. A typical use-case is the dynamic choice of an implementation (based on a core-language value):

```
module type Map = ...
let m1 = (module ListMap : Map)  (* core-language value *)
let m2 = (module HashMap : Map)  (* core-language value *)
let n = ... (* computation *)

module G = (val (if n < threshold then m1 else m2) : Map)
```

```
val m1 : (module Map)
val m2 : (module Map)
module G : Map
```

The modules `ListMap` and `HashMap` are turned into core-language values `m1` and `m2` with the construct `(module M : S)`, where the signature `S` must be provided (and, in OCaml, must be a module-type name). Conversely, first-class modules are turned back into modules with the construct `(val m : S)`, where, again, the signature `S` must be provided (and be a module-type name). A new core-language type of the from `(module S)` is added.

The other common pattern is to use local modules to *unpack* a first-class module argument inside a function, as in:

```
let f (m : (module Map)) =
  let module M = (val m) in
  ... (* use M *)
```

The types of first class modules are associated with a new form of subtyping in the core-language: `(module S1)` can be a subtype of `(module S2)` even if `S1` and `S2` are not (syntactically) equal. The signatures must be mutual subtypes and there should be no reordering of value field (types and module types fields can be reordered). This is done to ensure that there is a *code-free* coercion from a module of signature `S1` to a module of signature `S2`: a module of signature `S1` could be seen as `S2` without changing its dynamic representation. Hence, we have for instance:

```
module type S1 = sig type t = int val x : t   val y : bool              end
module type S2 = sig              val x : int val y : bool type t = int end

let f : (module S1) → (module S2) = fun x → x
```

```
val f : (module S1) → (module S2) = <fun>
```

[3]This remains true with first-class modules, as they are associated with a named module-type, not an inferred signature.

[4]Sometimes called *packaged modules*.

Overall, first-class modules are a relatively small extension of the language, only requiring two new constructs and the introduction of a new core-language type, but they are quite powerful. As presented in Yallop and Kiselyov [2010], they allow to encode existential types, polymorphic recursion, GADTs, and some form of generic programming.

Modular explicits/implicits Yet, first-class modules are limited, as (1) types coming from first-class module arguments cannot appear in the resulting type of functions and (2) they do not permit polymorphism over abstract types of module parameters of core-language functions. One could say "taking a first-class module as input does not make one a first-class functor". Overall, it means that first-class modules cannot be used as (possibly implicit) type-class parameters, like it is done in Scala [Oliveira et al., 2010], while Dreyer et al. [2007] have shown that ML modules could encode Haskell-like type-classes. To lift those restriction, a new feature called *modular implicits* was proposed by White et al. [2014]: module arguments to core-language functions could be automatically inferred from a base of implicit modules and functors. This would provide a (controlled) type-classes-like mechanism, with, among other things, ad-hoc polymorphism. To represent the module arguments that can appear in the domain and the codomain of function types, the language requires *module-dependent types* and *first-class functors*, which is often summed up as *modular explicits* [Vivien et al., 2024]. Modular implicit is currently being researched, as it poses both theoretical and practical challenges. One key challenge of inference is to ensure that the solution is unique, up to a certain notion of module equivalence. Here, it would make sense to reuse the notion of module equivalence based on aliasing, as discussed in Section 2.2.4. Overall, it would undoubtedly expand the interaction between the core and module languages in a powerful way, and make for a great extension to OCaml. In the language Scala, the implicit arguments (and implicit coercions) have become a widely used feature of the language [Křikava et al., 2019] (98% of the studied Scala projects use implicits, and 78% define new implicits).

1ML The stratification between the core and module language can even be questioned altogether: is it an historical artifact or is technically justified? An interesting stance is taken by Rossberg [2018], who shows that the core and module languages can be unified in a single language without (significant) loss of expressivity or inference. In 1ML, modules and core-language values co-exist, and are distinguished at the type level by a light form of predicativity. It has been extended with recursivity [Jaakkola, 2020]. This approach is appealing and could be the base of a new language. More research is needed to see if it could scale to all the features of ML modules. In practice, the stratification between core and module languages has also benefits regarding maintenance, as it (relatively) separates the features from the two languages levels, which makes it easier to work on one without having to worry about the other.

- C **First-class modules**: allow the core-language to handle modules, and the module language to depend on core-language values.
- C **Local modules**: allow an interleaving of core and module definitions. Especially useful to unpack a first-class module inside a value definition.
- C **Modular explicits**: enable *module-dependent* types and module-polymorphic functions (a.k.a. first-class functors).
- C **Modular implicits**: provide a automatic resolution of module arguments, with the convenience of type-classes without restricting the module language.

6.2.2 Extending the signature language

In this section we present some extensions of the signature language. First, we consider *signature modifiers*: their key objective is to allow the user to modify and compose existing

signatures without having to rewrite them from scratch. In practice, it is essential as rewriting whole signatures would be extremely cumbersome and verbose, and it allows for more sharing of signatures within the typechecker. Second, we discuss *parametric signatures*.

Modifying signatures A very complete overview of all features for handling signatures can be found in Ramsey et al. [2005]. The ones present in OCaml are:

- **Flat composition**: similarly to module composition, one can import all the fields of a structural signature inside another one with the **include** operator:

```
module type T = sig type t val x : t end
module type T' = sig include T val y : t end
```

```
module type T' = sig type t val x : t val y : t end
```

- **Substitutions**: one can change an abstract type field into a concrete one by using the **with type** t = ... construct:

```
module type T = (sig type t val x : t end) with type t = A of int | B of bool
```

```
module type T = sig type t = int val x : t end
```

  Similarly, one can change an abstract *module-type* field into a concrete one by using the **with module type** construct, and one can add aliasing information by using the **with module** construct.

- **Destructive substitutions**: one can replace a type field by a manifest type definition and remove the corresponding field, using the **with type** t := ... (note the ":=" instead of "="):

```
module type T = (sig type t val x : t end) with type t := int
```

```
module type T = sig val x : int end
```

  Similarly, one can do a destructive substitution of a module-type field (using the construct **with module type** T := S) and replace a module binding with an alias (using the construct **with module** X := P).

- **Local definitions**: one can define local aliases for types inside a signature with the declaration **type** t := ..., as in:

```
module type T = sig type t := int * int val x : t * t end
```

```
module type T = sig val x : (int * int) * (int * int) end
```

  The local alias can be used to write the rest of the signature but is then inlined (it does not appear in the resulting signature). The construct also exists for module types and aliasing information.

Subsuming and completing those, Ramsey et al. [2005] proposes a complete set of constructs to (1) change translucency (adding or removing type equalities), (2) add, remove or change fields, and (3) define local short names inside a signature. For completeness, we also mention the proposal[5] for a *strengthening* construct S **with** P that rewrites all abstract type fields to point to the same field in the module at path P.

Overall, the key design point from a language perspective is not how many constructs are provided, but rather which ones are kept in the internal representation of types and which ones are elaborated away [6]. The main motivation behind the proposal for lazy strengthening

[5] https://github.com/ocaml-flambda/flambda-backend/pull/1337

[6] Some constructs are just syntactic sugar, but most require a context-sensitive rewriting of the signature. That's why we say that those are *elaborated* away.

was the possibility of sharing signature: instead of duplicating a (possibly large) signature, only the "diff" is stored.

Parametric signatures Just as parametric types can be useful in the core language, one can want *parametric signatures*: i.e., signatures that take as input a type (or module-type) parameter. The substitution mechanism presented above can be used to "mimic" parametric signatures. Indeed, one can add an abstract type field for each parameter and "instantiate" with destructive substitutions (which effectively removes the corresponding type field). For instance:

```
module type T = sig
  type t (* used as a parameter *)
  type u = t * t
  val x : u
end
module X : (T with type t := int) = ...
```

```
module X : sig type u = int * int val x : u end
```

However, this approach has two downsides: it uses a non α-convertible binding (the type field t has a fixed name) and does not rely on kinds to force the signature to be used for a module only when fully instantiated. Instead, we advocate for truly parametric signatures, which could look like:

```
module type 'a T = sig type u = 'a * 'a val x : u end
```

The type-system would then recognize `T` as being *of a higher kind*: no module can have `T` as a signature, only instances of `T`, of the form `(int T)`, `(bool T)`, etc. However, it should be noted that we have not explored the interaction between parametric signatures and abstract signatures, and we fear that it might be problematic[7]. Besides, parametric signatures that take higher-order types as parameters would lead to the higher-order unification problem during subtyping.

- F **Signature manipulation**: following Ramsey et al. [2005], we advocate for an *expressive language of signatures*. Some constructs should be elaborated away, while some should be kept (as long as possible) throughout the typechecking process, to increase sharing of signatures inside the typechecker.
- C **Parametric signatures**: useful in practice, they should be restricted to maintain decidability of subtyping.

6.2.3 Interacting with the inference of signatures

Up until now, the interaction between the user and the type-system were binary: either the user accepts the inferred signature, or it forces a user-written one via ascription. Yet, there are use-cases for a richer interaction.

Using the result of inference One can get the result of the inference of any module expression by using the construct **`module type of`**, as in:

```
module X = struct type t = int end
module F (Y : sig type t end) = struct type u = Y.t * bool end
module type T = module type of F(X)
```

```
module type T = sig type u = int * bool end
```

[7] Our treatment of abstract signatures in Section 4.5 only supports non-parametric signatures.

This effectively injects module expressions inside signatures, as one can get the module type of any module expression. In OCaml, there is a special case for module expressions that "introduce" abstract types: **module type of** then returns a signature with abstract type fields, while it otherwise returns a signature with concrete type fields. For instance, we have:

```
module X  = struct type t end (* abstract type binding *)
module X' = X

module type T  = module type of X
module type T' = module type of X'
```

```
module type T  = sig type t end
module type T' = sig type t = X.t end
```

This behavior can be understood syntactically, but makes the construct very sensible to the order of definitions. Indeed, it gives a special role to some paths, and breaks some expected properties of the construct. Notably, **module type of** is not the inverse of ascription: for a module expression `M`, doing an ascription with the module type of itself can remove type equalities: `(M:module type of M)` does not have the same signature as `M`. For instance, we have:

```
module X = struct type t end
module X1 = X
module X2 = (X: module type of X)
```

```
module X : sig type t end
module X1 : sig type t = X.t end
module X2 : sig type t end (* lost type equality ! *)
```

In our quest for regularity, we think the construct should be made simpler by always returning the signature with all type equalities (the *strengthened* signature). In the above example, we should have **module type of** `X` resolved to **sig type** `t = X.t` **end**. The user could then use the signature modifiers of Section 6.2.2 to make some type fields abstract if needed.

Tweaking inference Sometimes, the result of the inference is *almost* the desired signature. Yet, in order to get exactly the desired signature, the user has to use an explicit ascription. This is cumbersome, especially at the top-level of a file. Instead, we advocate for constructs to change type declarations in the inferred signature, by making them abstract or private. This could look like:

```
module M = struct
  type t = int [export as abstract]
  type v = bool [export as private]
end
```

```
module M : sig type t type v = private bool end
```

Removing fields from the resulting signature can already be achieved with **open**. Another option is to reuse the signature modifiers of Ramsey et al. [2005], and, as they suggest in section 6.2, extend them to act on the inferred signature of a module.

Overall, we have the following features

- C **Getting the result of inference**: the user should be able to get the signature of any module expression that does not introduce new abstract types. This signature should be strengthened to have no abstract type field.
- C **Inference tweakers**: the user should be able to change the result of inference without having to use an explicit ascription.

Chapter 7

Conclusion

We have presented an overview of the main challenges of ML modules, and, to tackle those challenges, two type systems: M^ω and ZipML. Those type systems allowed us to explore different ways of representing abstraction and scopes, with similar expressivity but different practical trade-offs. A question remains: *is one better than the other*? Overall, we think they serve different purposes. M^ω is *standardized*, as it uses standard F^ω type-sharing mechanisms that are easy to understand. Once the intellectual effort of having a specification by elaboration is digested, it allows one to build a logical intuition of the typechecking of modules. The only "difficult" point that remains is extrusion. It is well suited to serve as a (relatively) simple model of a module language for other formalization efforts aimed at specifying OCaml (or other ML dialects). Overall, I like to think of M^ω as a *general purpose*, standard module system.

By contrast, ZipML can be seen as a *specialized* module system, designed for modeling a real-world implementation. ZipML embraces a natural intuition for type-systems: the internal representation of types for typechecking should be the same as the external language of types. It assumes the added complexity that comes with this choice: a practical language for writing signatures is not necessarily practical for a typechecker.

Despite a significant difference in presentation, the two systems also have a lot in common. At a high-level, both rely on a shared idea: once introduced, abstract types will survive modifications of the signature, regardless of whether or not an anchoring point is visible. ZipML implements this idea lazily, introducing floating fields on demand (when projecting), while M^ω introduces abstract type variables eagerly.

Future works

In this section we discuss future works and perspectives.

Missing features

If we look back at the wish-list established throughout Chapter 2, there are two main features that have been left out of this work: recursive composition and modular implicits. The latter is more of a core-language feature and could probably fit well with either M^ω or ZipML, as it mostly relies on module paths and a good criterion for module equivalence, which are treated similarly in both type systems.

Recursive composition, however, is a key difficult feature that is covered by neither M^ω nor ZipML. While some works have paved the way for general recursive composition, culminating in the Rossberg and Dreyer [2013] paper, future research is needed to understand the possible interaction with applicative functors (especially when sealing inside applicative functors). We believe that there is an unexplored sweet spot where recursive and sequential composition coexists, such that users can benefit from the best of both worlds: the simplicity of sequential composition for most use-cases, and the flexibility of recursive composition for advanced ones.

Finally, in both M^ω and ZipML, we adopted a strict, *no-loss of type-sharing* criterion for anchoring in M^ω and for the simplification of zippers in ZipML. However, as pointed out in Section 4.6, there seems to be *harmless* losses of type-sharing: cases where loosing type

equalities is not observable by the module language. Characterizing those cases and proving that they are indeed harmless would constitute interesting and novel research.

Implementation

The obvious follow-up of this work is to implement M^ω and ZipML as typecheckers. After proposing M^ω, the feedback from the OCaml community was mixed: while interesting, the design was not seen a short or mid-term replacement for the module part of the typechecker, notably because it departed from the current syntactic approach quite significantly. It was considered as a long-term (and thus, uncertain) proposal. The key concern was that switching to an M^ω-based solution would drastically affect the performance of the typechecker, and also require technical changes to the core language (especially if qualified type names were to be removed in favor of abstract type variables).

This mixed feedback played a role in motivating the research effort that led to ZipML. Benefiting from the insights of M^ω and the technical novelty of zipper signatures, ZipML was designed with two main constraints in mind from the beginning: maintaining performance of typechecking and limiting the interaction with the core-language to its absolute minimum. While the former is achieved by preventing duplication and inlining of signatures, the latter was obtained by making the notion of type declaration as general as possible, only requiring knowing if a type is inline-able or not.

We believe that, while M^ω could be the base of a new typechecker, ZipML could be implemented in current OCaml. Such endeavor would be approached by successive steps. First, implementing transparent signatures and differentiation between static and dynamic aliases. Then, zipper signatures could be added, with a trivial simplification algorithm that only garbage collects and inlines floating fields. This would already allow for a delayed avoidance. Finally, the full-fledged simplification algorithm could be added.

Mechanization

This work could benefit from mechanization in several ways:

- **Soundness of a module system with sealing inside applicative functors**: Mechanizing the soundness proof of the generative subset would not be of much interest, as it has already been done in *F-ing* (Rossberg et al. [2014]). However, using transparent existentials in a mechanized proof of soundness of the applicative functors have not been done. We hope that our claims regarding proof factorization between applicative and generative cases would materialize. It would also be insightful to compare such proofs with the ones of Crary [2020], who uses a system based on singleton kinds.

- **Semantic model**: Type safety of F^ω is not sufficient to prove *abstraction safety*. While our identity tags or the phantom types of *F-ing* (Rossberg et al. [2014]) hints at abstraction safety, it does not provide a satisfactory theorem. Indeed, we would like to be able to *attach* to an abstract type an arbitrary predicate, and show that, as soon as it is validated by the functions and values of the module, it cannot be invalidated later on. We believe that building a semantic model of M^ω types could help conducting such proof.

- **Interoperability**: Besides the gained confidence, providing a standardized and reusable mechanized proof of soundness for the module language could be very useful to other formalization efforts aimed at proving OCaml code – a topic of quickly growing importance.

Bridging the gap with other languages

This work was done with the goal of improving the situation of ML-modules in mind, hoping that it would also serve other languages in the long run. The example of Scala has shown that the power of ML modules can inspire other designs outside of its community. We think exploring further how the features presented in this thesis can be imported in other contexts is an interesting and impactful research topic.

User studies

One significant challenge in the discussion about the redesign of ML modules is the lack of user studies. Comparing SML and OCaml alone is insufficient to evaluate the impact of applicative functors, for instance. This *blind spot* – designing languages based primarily on theoretical principles while insufficiently incorporating user-feedback – is not specific to module systems. However, the nature of modular programming exacerbates this issue, as the advantages and drawbacks of language design become apparent only at larger scales, and having users write large modular programs for a study seems a lot to ask. I believe that finding new ways to embrace the human-psychological and sociological aspects of programming in the design of languages constitutes interesting and under-explored research.

Bibliography

Sandip K. Biswas. 1995. Higher-Order Functors with Transparent Signatures. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/199448.199478

Clément Blaudeau and Fengyun Liu. 2022. A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the Celsius model. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 151 (Oct. 2022), 29 pages. https://doi.org/10.1145/3563314

Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2024. Fulfilling OCaml Modules with Transparency. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 101 (apr 2024), 29 pages. https://doi.org/10.1145/3649818

Gérard Boudol. 2004. The recursive record semantics of objects revisited. *Journal of Functional Programming* 14, 3 (2004), 263–315. https://doi.org/10.1017/S0956796803004775

Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30 (2020), e24. https://doi.org/10.1017/S0956796820000222

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module? *SIGPLAN Not.* 34, 5 (may 1999), 50–63. https://doi.org/10.1145/301631.301641

Derek Dreyer. 2007a. Recursive type generativity. *Journal of Functional Programming* 17, 4-5 (2007), 433–471. https://doi.org/10.1017/S0956796807006429

Derek Dreyer. 2007b. A type system for recursive modules. *SIGPLAN Not.* 42, 9 (oct 2007), 289–302. https://doi.org/10.1145/1291220.1291196

Derek Dreyer, Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 236–249. https://doi.org/10.1145/604131.604151

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular Type Classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 63–70. https://doi.org/10.1145/1190216.1190229 event-place: Nice, France.

Derek Dreyer, Robert Harper, and Karl Crary. 2005. *Understanding and evolving the ML module system*. Ph. D. Dissertation. USA. AAI3166274.

Jacques Garrigue and Didier Rémy. 2012. Tracing ambiguity in GADT type inference. (June 2012).

Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

Robert Harper, Robin Milner, and Mads Tofte. 1987. A type discipline for program modules. In *TAPSOFT '87*, Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–319.

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/96709.96744

Robert Harper and Christopher A. Stone. 2000. A type-theoretic interpretation of standard ML. In *Proof, Language, and Interaction*. https://api.semanticscholar.org/CorpusID:9208816

Tom Hirschowitz and Xavier Leroy. 2005. Mixin modules in a call-by-value setting. *ACM Trans. Program. Lang. Syst.* 27, 5 (Sept. 2005), 857–881. https://doi.org/10.1145/1086642.1086644

GÉRARD HUET. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

Hyeonseung Im, Keiko Nakata, Jacques Garrigue, and Sungwoo Park. 2011. A syntactic type system for recursive modules. *SIGPLAN Not.* 46, 10 (oct 2011), 993–1012. https://doi.org/10.1145/2076021.2048141

Pauli Jaakkola. 2020. *A Type System for First-Class Recursive ML Modules*. Master's thesis. https://trepo.tuni.fi/bitstream/handle/10024/123958/JaakkolaPauli.pdf

Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 163 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360589

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 142–153. https://doi.org/10.1145/199448.199476

Xavier Leroy. 2000. A modular module system. *J. Funct. Program.* 10, 3 (2000), 269–303. http://journals.cambridge.org/action/displayAbstract?aid=54525

Xavier Leroy. 2003. A proposal for recursive modules in Objective Caml. *Available from the author's website* (2003).

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. The OCaml system. https://ocaml.org/manual/index.html

Runhang Li and Jeremy Yallop. 2017. Extending OCaml's 'open'. In *Proceedings ML Family / OCaml Users and Developers workshops, ML/OCaml 2017, Oxford, UK, 7th September 2017 (EPTCS, Vol. 294)*, Sam Lindley and Gabriel Scherer (Eds.). 1–14. https://doi.org/10.4204/EPTCS.294.1

Fengyun Liu. 2020. Safe initialization of objects. (2020), 168. https://doi.org/10.5075/epfl-thesis-8265

Fengyun Liu, Ondřej Lhoták, David Hua, and Enze Xing. 2023. Initializing Global Objects: Time and Order. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 268 (Oct. 2023), 28 pages. https://doi.org/10.1145/3622844

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. 4 (2020), 1–28. Issue OOPSLA. https://doi.org/10.1145/3428243

Fengyun Liu, Ondřej Lhoták, Enze Xing, and Nguyen Cao Pham. 2021. Safe object initialization, abstractly. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. Association for Computing Machinery, 33–43. https://doi.org/10.1145/3486610.3486895

Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (aug 2024), 30 pages. https://doi.org/10.1145/3674642

Robin Milner. 1972. Implementation and applications of Scott's logic for computable functions. *SIGPLAN Not.* 7, 1 (Jan. 1972), 1–6. https://doi.org/10.1145/942578.807067

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, MA, USA.

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/2319.003.0001

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

John C. Mitchell and Gordon D. Plotkin. 1985. Abstract Types Have Existential Types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 37–51. https://doi.org/10.1145/318593.318606

Jeffrey C. Mogul. 2006. Emergent (mis)behavior vs. complex software systems. *SIGOPS Oper. Syst. Rev.* 40, 4 (April 2006), 293–304. https://doi.org/10.1145/1218063.1217964

Benoît Montagu. 2010. *Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types. (Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts)*. PhD Thesis. École Polytechnique, Palaiseau, France. https://tel.archives-ouvertes.fr/tel-00550331

Benoît Montagu and Didier Rémy. 2009. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. Savannah, GA, USA, 354–365. https://doi.org/10.1145/1480881.1480926

Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. (2006), 13.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. *SIGPLAN Not.* 45, 10 (Oct. 2010), 341–360. https://doi.org/10.1145/1932682.1869489

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages.* The MIT Press.

Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 53–65. https://doi.org/10.1145/1480881.1480890

Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005. An expressive language of signatures. *SIGPLAN Not.* 40, 9 (Sept. 2005), 27–40. https://doi.org/10.1145/1090189.1086371

Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. 2021. A practical mode system for recursive definitions. 5 (2021), 45:1–45:29. Issue POPL. https://doi.org/10.1145/3434326

Andreas Rossberg. 2006. The missing link: dynamic components for ML. *SIGPLAN Not.* 41, 9 (sep 2006), 99–110. https://doi.org/10.1145/1160074.1159816

Andreas Rossberg. 2018. 1ML - Core and modules united. *J. Funct. Program.* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205

Andreas Rossberg and Derek Dreyer. 2013. Mixin'Up the ML Module System. *ACM Trans. Program. Lang. Syst.* 35, 1 (April 2013), 2:1–2:84. https://doi.org/10.1145/2450136.2450137

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (Sept. 2014), 529–607. https://doi.org/10.1017/S0956796814000264

Claudio V. Russo. 1996. Standard ML Type Generativity as Existential Quantification. https://api.semanticscholar.org/CorpusID:14337913

Claudio V. Russo. 2000. First-Class Structures for Standard ML. *Nord. J. Comput.* 7, 4 (2000), 348–374.

Claudio V. Russo. 2001. Recursive structures for standard ML. *SIGPLAN Not.* 36, 10 (oct 2001), 50–61. https://doi.org/10.1145/507669.507644

Claudio V. Russo. 2004. Types for Modules. *Electronic Notes in Theoretical Computer Science* 60 (2004), 3–421. https://doi.org/10.1016/S1571-0661(05)82621-0

Didier Rémy and Jérôme Vouillon. 1997. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages.* Paris, France, 40–53.

Didier Rémy and Jérôme Vouillon. 1998. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems* 4, 1 (1998), 27–50.

Chung-Chieh Shan. 2004. Higher-order modules in System F^{ω} and Haskell. (01 2004).

Zhong Shao. 1999. Transparent Modules with Fully Syntactic Signatures. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.* ACM, 220–232. https://doi.org/10.1145/317636.317801

Alexander J. Summers and Peter Mueller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011-10-22) *(OOPSLA '11)*. Association for Computing Machinery, 1013–1032. https://doi.org/10.1145/2048066.2048142

Don Syme. 2006. Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge. *Electron. Notes Theor. Comput. Sci.* 148, 2 (2006), 3–25. https://doi.org/10.1016/j.entcs.2005.11.038

Mads Tofte. 1990. Type inference for polymorphic references. *Inf. Comput.* 89, 1 (Sept. 1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

Samuel Vivien, Didier Rémy, Thomas Réfis, and Gabriel Scherer. 2024. On the design and implementation of Modular Explicits. (Sept. 2024). https://cambium.inria.fr/~remy/ocamod/implicits.html Presented at the OCaml 2024 workshop, available electronically.

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*. 22–63. https://doi.org/10.4204/EPTCS.198.2

Jeremy Yallop and Oleg Kiselyov. 2010. First-class modules: hidden power and tantalizing promises. (2010).