Managing Logical and Computational Complexity using Program Transformations

Nicolas Tabareau

École Doctorale STIM
UMR CNRS 6241 LINA

UNIVERSITÉ DE NANTES

Habilitation à Diriger les Recherches

Managing Logical and Computational Complexity using Program Transformations

Rendre Compte de la Complexité Logique et Calculatoire à travers des Transformations de Programmes

présentée et soutenue publiquement par

Nicolas Tabareau

le 24 novembre 2016

devant le jury composé de

Thierry	Coquand	Göteborg University (examinateur)
Hugo	Herbelin	Inria Paris (rapporteur)
Daniel	Hirschkoff	Ens Lyon (rapporteur)
Claude	Jard	Université de Nantes (examinateur)
Alan	Schmitt	Inria Rennes Bretagne Atlantique (rapporteur)

Table of Contents

Remerciements

Mains aux fleurs, Picasso (1958)

Je voulais dire *merci* aux membres du jury. Qu'ils sachent que je suis très honoré de leur présence à ma soutenance.

Je voulais dire *merci* à mes collègues, co-auteurs, étudiants ou post-docs, pour ces moments de partage et d'envolées scientifiques qui font de ce métier toute sa spécificité.

Je voulais dire *merci* à mes filles, pour leur spontanéité qui fait oublier en un clin d'œil toute contrariété.

Je voulais dire *merci* à ma princesse, pour sa fraîcheur toujours présente après ces 17 années qui nous unient.

Introduction

Le saut dans le vide,
Klein (1960)

Contents

Une Introspection Rétrospective

Remarque 1

Bien que le reste de ce manuscrit soit écrit de manière conventionnelle, cette section est écrite à la première personne pour souligner le fait qu'elle développe un point de vue personnel.

La Sémantique de la Compilation. Durant mon stage au sein de Microsoft Research à Cambridge en 2007, j'ai commencé à étudier la sémantique de la compilation des langages de programmation [BT09] sous la délicieuse supervision de Nick Benton. Mes intérêts scientifiques de l'époque portaient principalement sur la sémantique dénotationelle des langages de programmation, mais j'ai alors commencé à réaliser[1] que la compilation—ou plutôt la traduction—d'un programme écrit dans un langage complexe de haut niveau vers un langage plus simple était un moyen très primitif et efficace pour donner un sens au programme complexe.

[1] J'ai aussi commencé à réaliser que j'aimais beaucoup l'assistant de preuve Coq [CDT15], ce qui devrait être clair à la lecture du reste de ce manuscrit.

La Programmation par Aspects. Cette remarque m'a poursuivi alors que je commençais à étudier la sémantique de la programmation par aspects lors de mon arrivée dans l'équipe Ascola (Aspects Components Languages) en 2009. La programmation par aspects [KLM+97] est un paradigme de programmation qui a pour but d'accroître la modularité du code en permettant la séparation des propriétés transverses. Nous appelons propriété transverse une propriété qui ne peut pas être définie en utilisant les mécanismes de modularité classiques, comme par exemple les classes, les modules ou les composants. Le calcul de la trace d'un programme ou les problématiques de sécurité forment deux exemples de propriétés transverses, et sont donc sujettes à éparpillement et intrication. L'idée de la programmation par aspects est de ramener toutes les parties du code qui traitent d'une propriété transverse dans un même endroit du programme, à travers l'utilisation d'un nouveau type de module, appelé un *aspect*. Les deux principaux constituants d'un aspect sont les *coupes* (pointcuts en anglais) et les *advices*[2] Une coupe sélectionne les points d'exécution d'intérêt du programme de base, par exemple, les appels de méthode dans le cas du calcul de trace. Un advice définit ce qui doit être fait lorsqu'un point d'exécution est sélectionné, typiquement l'affichage d'un message avant et/ou après un appel dans le cas de l'affichage de la trace d'un programme. Le processus qui consiste à composer des aspects avec un programme de base, afin d'obtenir une implémentation concrète du programme tel qu'il aurait été écrit sans aspect, s'appelle le *tissage*.

Mon premier travail sur la sémantique d'un calcul distribué avec aspects [Tab10] a été obtenu en traduisant ce calcul vers le "join calculus" [FG96a], considérant ainsi le tissage comme une transformation de programme. J'ai ensuite étudié la sémantique de la programmation par aspects pour des langages fonctionnels [Tab11, Tab12], à la lumière de la traduction monadique, qui constitue une transformation de programme bien connue dans le cadre de la programmation fonctionnelle. Les monades [Mog91] fournissent un mécanisme pour embarquer et raisonner sur des effets comme les états mémoires, les entrées/sorties ou la gestion des exceptions, et ce dans un cadre purement fonctionnel comme dans Haskell. Définir de la sémantique d'un langage fonctionnel avec aspects en utilisant une monade permet d'être modulaire sur les différentes sémantiques possibles des aspects (*i.e.*, le déploiement d'aspects, la gestion des interférences entre aspects), et permet aussi une implémentation directe dans le langage, comme nous l'avons montré sur Haskell [TFT13, FTT14b, FSTT14].

En utilisant des idées venant de la notion de portée pour les aspects, j'ai développé, avec Ismael Figueroa et Éric Tanter, une définition monadique de la notion de capacités pour les effets [FTT14a, FTT15] dans Haskell, capacités qui permettent d'imposer certaines politiques de sécurité sur les effets qui sont utilisés dans un programme Haskell. En plus du cadre monadique, ce travail utilise crucialement le mécanisme de type classes de Haskell pour décider statiquement du niveau de permission en se basant sur une implémentation du treillis des permissions avec des type classes.

Le Forcing en Théorie des Types. Entre temps, j'utilisais de plus en plus Coq pour formaliser certaines parties des preuves que je faisais sur papier. Cela me fit prendre conscience que même si la théorie des types (ou plus précisément le Calcul des Constructions Inductives, la théorie derrière Coq) est déjà très expressive, il manque cruellement la possibilité d'augmenter son contenu logique ou calculatoire facilement. Cela m'a amené à développer avec Guilhem Jaber et Matthieu Sozeau une version du forcing pour la théorie des types [JTS12]. Le *forcing* est une méthode originellement développée par Paul Cohen pour prouver l'indépendance de l'hypothèse du continu avec la théorie des ensembles

[2] Je me permets cette francisation maladroite du terme "advice" utilisé dans la programmation par aspects qui constitue déjà une utilisation libérale du terme anglais.

ZFC [Coh66]. L'idée principale est d'ajouter de nouveaux éléments à la théorie qui peuvent être *approximés* dans le système initial, en utilisant ce qu'on appelle des *conditions de forcing*. Pour adapter cette idée à la théorie des types, plutôt que d'utiliser l'approche habituelle du forcing à travers la théorie des ensembles, nous avons utilisé sa reformulation en terme de théorie des catégories que Bill Lawvere et Myles Tierney [Tie72] ont développé en utilisant le topos des faisceaux. Récemment, les travaux de Jean-Louis Krivine [Kri11] et Alexandre Miquel [Miq11a] ont montré que les techniques liées au forcing sont d'un grand intérêt pour l'extension de l'isomorphisme de Curry-Howard. Le point de départ de notre travail a été de connecter ces deux observations :

> "Le forcing intuitionniste en théorie des types correspond à l'internalisation de la théorie des préfaisceaux en théorie des types."

L'idée est d'étendre une théorie des types de base avec de nouveaux principes, obtenant ainsi une nouvelle théorie, appelée *couche de forcing*. Les termes et types de la couche de forcing peuvent être traduit vers la théorie de base en utilisant la *traduction de forcing*. La couche de forcing peut satisfaire de nouveaux principes logiques ou calculatoires, *e.g.*, l'existence de points fixes généraux lorsqu'on force avec les entiers naturels comme conditions de forcing—ce qui correspond à travailler avec le topos des arbres.

Plus récemment, nous avons réalisé qu'il manquait certaines propriétés calculatoires à notre traduction de forcing pour que celle-ci soit complètement utilisable en pratique, à cause de problème de cohérence. Voilà pourquoi nous avons changé notre fusil d'épaule en passant d'une interprétation en appel par valeur à une interprétation en appel par nom, sous l'initiative de Pierre-Marie Pédrot. Ce glissement nous a permis de donner une nouvelle version de la traduction de forcing [JLP+16] directement implémentable dans un module d'extension (alias plugin) de Coq. En effet, notre traduction est directement implémentable car elle interprète la conversion—*i.e.*, le fait que deux termes soient convertibles en utilisant l'égalité définitionnelle—par la conversion elle-même. Ceci est à opposer au fait que, dans la plupart des autres travaux sur le sujet et en particulier au niveau des modèles de la théories des types, la conversion est interprétée par l'égalité propositionnelle ou par une égalité externe fournie par le modèle. Dans ces cas, il se pose le fameux problème de cohérence car la règle de conversion se retrouve interprétée par un transport explicite d'un terme à l'autre, et ne se trouve plus implicite dans le terme de preuve. Bien que ce problème de cohérence ait été assez largement étudié, en particulier dans le travail récent de Pierre-Louis Curien, Richard Garner et Martin Hofmann [CGH14], il n'y pas de solution à ce jour pour produire une transformation de programme correcte de la théorie des types pour de tels modèles/transformations.

Cette observation m'amène à avancer la devise suivante, qui constitue une des clés de voûte de mon programme de recherche développée dans les pages qui suivent:

> "Une transformation de programme correcte et implémentable doit préserver la conversion de type de manière exacte."

Plus précisément, je pense qu'il est possible de modifier les transformations existantes pour qu'elles préservent la conversion, comme nous l'avons fait pour le forcing en passant d'une interprétation en appel par valeur à une interprétation en appel par nom.

Interopérabilité Dépendante. Une autre réflexion, qui me vint à l'esprit en discutant avec Éric Tanter sur l'approche graduelle pour les systèmes de types dans les langages de programmation, est qu'il manque à Coq un mécanisme similaire pour permettre une phase d'apprentissage moins abrupte pour les développeurs qui souhaitent passer d'un

cadre simplement typé à une cadre avec des types dépendants.[3] En fait, nous avons réalisé que le problème se pose aussi dans d'autres situations, lorsque par exemple on souhaite transformer un programme écrit en Coq en un programme écrit dans un langage simplement typé comme OCaml ou Haskell—un processus qui est appelé *extraction de programme*. Dans ce cas, l'absence de gradualité engendre une perte importante de propriétés entre le programme initial et le programme extrait. C'est avec ces problèmes en tête que j'ai travaillé avec Pierre-Évariste Dagand et Éric Tanter sur l'interopérabilité dépendante [TT15, DTT16] entre Coq et OCaml/Haskell, interopérabilité qui peut être vue comme un moyen automatique d'améliorer l'extraction de programmes, *i.e.*, la transformation de programme de Coq vers un langage simplement typé.

Aperçu du Reste du Manuscrit

Les pages qui suivent présentent mon programme de recherche pour les cinq années qui viennent, et peut-être plus.[4] Après ça, le reste du document est dédié à la présentation de trois de mes travaux dans des sujets qui sont assez différents mais qui partagent l'idée commune d'utiliser des transformations de programmes pour traiter de la complexité logique ou calculatoire. Tous ces travaux ont été publiés et sont rassemblés, standardisés et revisités ici dans le but de rendre ce manuscrit aussi indépendant que possible.

Le chapitre 1 reprend l'introduction dans la langue de Shakespeare, qui sera aussi la langue choisie pour présenter mes travaux de recherche.

Le chapitre 2 décrit une transformation de programmes écrits en Coq vers des programmes écrits en Coq, afin d'accroître le pouvoir logique et calculatoire de Coq. Ce chapitre est directement tiré du papier *The definitional side of the Forcing* [JLP+16] qui introduit le forcing dans sa présentation en appel par nom pour Coq. Ce travail est en collaboration avec Guihlem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot et Matthieu Sozeau.

Le chapitre 3 présente un calcul d'aspects distribué basé sur le join calcul. La complexité du déploiement des aspects et du tissage dans un cadre distribué est gérée en donnant une traduction directe de l'aspect join calcul vers le join calcul de base, ce qui donne à la fois une sémantique et une implémentation pour l'aspect join calcul. Je présente ici une version revisitée et étendue, avec la collaboration d'Éric Tanter, du papier *A theory of distributed aspects* [Tab10].

Le chapitre 4 décrit, dans une certaine mesure, une transformation de programme de Coq avec types dépendants vers Haskell ou OCaml, en faisant reposer des vérifications qui sont statiques dans Coq sur des vérifications dynamiques. Ce chapitre est tiré du papier *Partial Type Equivalences for Verified Dependent Interoperability* [DTT16]. C'est un travail commun avec Pierre-Évariste Dagand et Éric Tanter.

Ma Future Ligne de Recherche

L'égalité dans les assistants de preuve basés sur la théorie des types. Chaque année, les bogues logiciels coûtent des centaines de millions d'euros aux entreprises et administrations, comme en témoigne le bogue sur la division du processeur développée par Intel Pentium ou l'explosion au décollage de la fusée Ariane V. Il est donc logique que la notion de qualité du logiciel soit devenue de plus en plus présente, allant bien au-dela du cadre restreint des systèmes embarqués. Plus particulièrement, le développement

[3] J'utilise le terme "simplement typé" pour dire "sans types dépendants".

[4] La plupart des travaux à venir seront développés dans le cadre du projet *CoqHoTT*—ERC starting grant 637339.

d'outils pour construire des programmes qui respectent une spécification donnée est un des problèmes majeurs de la recherche actuelle et future en informatique. Les assistants de preuves basés sur la théorie des types, comm Coq [CDT15], équipé d'un mécanisme d'extraction qui produit du code certifié à partir d'une preuve, se retrouve ainsi de plus en plus dans la lumière. En effet, ils ont prouvé leur efficacité pour prouver la correction d'importants logiciels, comme le compilateur C du projet CompCert [Ler06]. Un des intérêts de l'utilisation de tels assistants de preuve du point de vue informatique est la possibilité d'extraire directement le code qui a été prouvé, et de l'exécuter comme n'importe quel logiciel. Par exemple, le compilateur de CompCert qui est extrait de sa certification est fiable et efficace, tournant seulement 15% moins que GCC 4 avec le niveau d'optimisation 2 (*i.e.*, gcc -02), un niveau d'optimisation qui est alors considéré comme non fiable même si efficace.

Malheureusement, la démocratisation de ces tels assistants de preuve souffre d'un handicap majeur, la discordance entre l'égalité telle que conçue en mathématique et l'égalité telle qu'implémentée dans la théorie des types. En effet, certains principes de base qui sont utilisés massivement et implicitement en mathématique—comme le principe d'extensionnalité propositionnelle de Church, qui stipule que deux formules sont égales lorqu'elles sont logiquement équivalentes—ne sont pas vérifiés en théorie des types. De manière plus problématique en informatique, le concept très naturel de considérer que deux fonctions sont égales lorsqu'elles le sont point à point n'est pas non plus vérifié en théorie des types, et doit ainsi être poser comme un axiome :

```
Axiom fun_ext : ∀ A B (f g : A → B), (∀ x, f x = g x) → f = g.
```

Bien sûr, ces principes sont consistants avec la théorie des types et les ajouter comme des axiomes est correct. Mais tous les développements qui en font appel dans la définition de programme produisent du code qui ne s'exécute pas, car il sera bloqué aux endroits où des axiomes ont été utilisés, car les axiomes sont des boîtes noires au niveau du calcul. Pour comprendre ce point de discorde, il nous faut regarder de manière plus précise comment est définie la notion d'égalité en théorie des types. Elle est définie en utilisant le type identité de Martin-Löf Id_A, qui vient avec une seule règle d'introduction (uniforme par rapport à A) :

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathrm{refl}_A\ x : \mathrm{Id}_A\ x\ x}$$

Cela signifie que le seul moyen de prouver que deux termes t et u sont égaux (ou plus précisément d'habiter le type $\mathrm{Id}_A\ t\ u$ sans hypothèse supplémentaire) est par réflexivité. Cela veut dire que les deux termes doivent être convertibles—au sens où ils ont la même forme normale modulo β-réduction (et quelques autres règles de réduction). D'une certaine manière, cette notion d'égalité est éminemment syntaxique. La raison pour laquelle le type identité est utilisé en théorie des types vient du fait qu'il est accompagné d'un principe d'élimination très puissant qui permet de substituer un terme par un terme qui lui est Id-équivalent n'importe où dans un type. C'est ce qu'on appelle le principe d'indiscernabilité des éléments identiques de Leibniz.

À l'opposé, la notion d'égalité en mathématique est éminemment sémantique, avec une définition qui est relative aux types des objets considérés. Cela explique pourquoi le principe d'extensionnalité des fonctions est gratuit dans ce cadre, car la définition de l'égalité est précisément construite sur l'égalité point à point pour le type des fonctions. Le problème avec ce point de vue est qu'il ne vient pas avec un principe général de substitution ce qui rend sa définition difficile à utiliser dans un assistant de preuve.

Ce point de discorde empêche d'utiliser des outils sémantiques plus élaborés dans le développement de preuves, car ils reposent sur la manipulation explicite de l'égalité mathématique, ainsi que sur l'utilisation d'axiomes. Bien sûr, des travaux importants ont été faits afin de contourner cette difficulté en définissant d'autres notions d'égalité que celle fournie par la théorie des types, en utilisant la notion de sétoïdes [BCP03]. Mais cela ne permet pas d'avoir une notion d'égalité comme les mathématiciens ont l'habitude d'utiliser au tableau, car le principe général de substitution n'est pas valide pour ces égalités. En effet, lorsqu'un développeur souhaite utiliser une notion d'égalité basée sur les sétoïdes et réécrire avec les égalités dans une expression, il doit prouver au préalable que cette expression valide le principe de substitution. Et lorsque cette expression provient d'un développement extérieur, cela peut s'avérer être une tâche très complexe, même pour une seule expression. Ce traitement non-uniforme de l'égalité pose aussi un problème lorsqu'il s'agit de composer des preuves entre elles, ce qui constitue pourtant un point fondamental du développement des mathématiques. Tous ces problèmes ont pour conséquence que même si des développements Coq conséquents ont vu le jour, ils ont tous été faits par des virtuoses capables de jongler avec des concepts avancés à la fois en informatique et en mathémaiique, ce qui exclut de facto un développeur ou mathématicien lambda. Mon projet de recherche a pour but de résoudre ce problème en fournissant un assistant de preuve dans lequel développeurs et mathématiciens pourront manipuler l'égalité comme ils le font habituellement au tableau.

La naissance de la théorie homotopique des types. Pour corriger cette dissonance fondamentale entre l'égalité en mathématique et l'égalité en théorie des types, Martin Hofmann et Thomas Streicher ont introduit un nouveau point de vue sur la théorie des types, dans lequel les types sont vus non pas simplement comme des ensembles, mais comme des ensembles munis d'une structure homotopique [HS96]. Ainsi, chaque type vient avec une notion d'égalité qui lui est propre et la structure homotopique décrit comment substituer un terme par un autre terme équivalent dans un type. Vladimir Voevodsky a reconnu récemment que cette interprétation homotopique des types satisfait de surcroît une propriété cruciale, appelée univalence, et qui n'avait jamais été considérée jusqu'à présent en théorie des types. Le principe d'univalence stipule que deux types `T` and `U` sont égaux si et seulement si ils sont équivalents (ou isomorphes)[5]:

```
Definition univalence := ∀ T U : Type, (T = U) ≅ (T ≅ U).
```

Le principe d'univalence subsume les autres principes manquant à l'égalité et constitue donc la clé pour équiper la théorie des types avec une notion d'égalité qui soit compatible avec les raisonnements faits traditionnellement en mathématique. Dans les développements récents autour de la théorie homotopique des types [Uni13], l'univalence est ajoutée à la théorie des types de Coq (ou Agda) à l'aide d'un nouvel axiome. Ce principe a des conséquences très importantes pour la formalisation de la théorie de l'homotopie, comme par exemple la preuve que le premier groupe d'homotopie de la sphère est équivalent à $\mathbb{Z}$ [LS13]. Mais le principe d'univalence a aussi des conséquences très intéressantes en informatique. Il permet d'utiliser une fonction/méthode d'une bibliothèque—qui a été définie pour un type de données `A`—sur un autre type de données `B`, dès lors qu'il y a un isomorphisme explicite entre `A` et `B`. Supposons par exemple que `A` soit le type inductif des listes.

```
Inductive list₁ A : Type := nil₁ : list₁ A | cons₁ : A → list₁ A → list₁ A.
```

[5] En fait, comme nous le définissons formellement en Section 2.6.3, ce principe dit plus précisément que la flèche canonique de `T = U` vers `T ≅ U` is an equivalence.

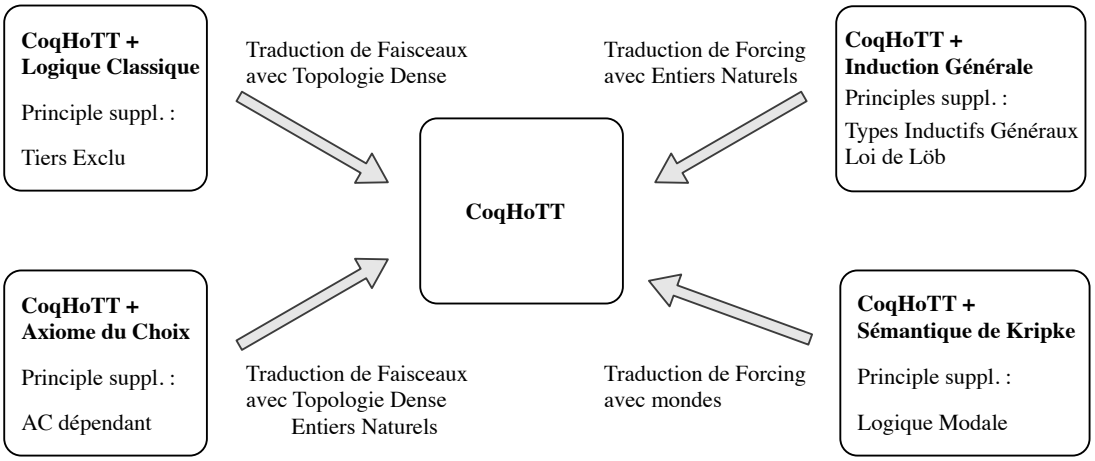


Figure 1: Extension de Coq en les transformations de programme venant de la logique mathématique

Prenons maintenant un développeur qui a utilisé son propre type inductif pour les listes au cours de son développement, et qui a, par convenance défini le constructeur `cons` en prenant les arguments dan le sens inverse de l'ordre habituel :

```
Inductive list₂ A : Type := nil₂ : list₂ A | cons₂ : list₂ A → A → list₂ A.
```

En utilisant l'univalence, peu importe la représentation utilisée pour les listes, il devient facile de passer d'un type à l'autre car les deux types sont isomorphes et peuvent être considérés comme égaux. Ce changement de base/type est très fréquent et peut être utilisé dans différents contextes: pour des raisons d'efficacité ou de simplicité d'un algorithme (comme par exemple pour les algorithmes de tri et les différentes représentations d'une liste), pour des raisons de compatibilité entre la représentation des données (*e.g.*, mètre ou pouce, la date au format américain ou européen)... L'univalence permet de faire une conversion automatique d'un programme agissant sur `A` à un programme agissant sur `B`, tout en ayant la garantie que les deux programmes produisent des résultats équivalents. Ceci peut être très utile dans des phases un peu plus avancées de développement logiciel. Mais comme la théorie homotopique des types est éminemment d'ordre supérieur, l'univalence peut aussi être utilisée pour replacer un programme par un autre programme, dès lors que ces deux programmes sont isomorphes, *i.e.*, indistinguables de l'extérieur. Cela forme donc un moyen de faire des optimisations prouvées correctes.

Objectifs pour les Cinq Prochaines Années

Malheureusement, postuler l'univalence par un axiome casse une propriété fondamentale des preuves obtenues avec un assistant de preuve basé sur la théorie des types: cela casse le mécanisme d'extraction qui permet d'obtenir un programme qui calcule à partir de la formalisation. Le point de départ de mon projet de recherche est de construire un système où l'univalence est intégrée nativement sans axiome. Ainsi, la théorie obtenue sera plus régulière, plus proche de la structure de topos (en fait ∞-topos), qui est une notion fondamentale en logique mathématique. Ce point est très important pour permettre de revisiter les transformations de modèles faites en logique, habituellement définies sur des topoï, vers des transformations agissant sur la théorie des types. Concrètement, il deviendra possible d'étendre de manière modulaire la logique gouvernant l'assitant de preuve, *e.g.*, en

ajoutant un principe d'induction général, les axiomes de la logique classique, l'axiome du choix ou les règles de la logique modale (Fig. 1).

> Le but principal de notre projet est de fournir une nouvelle génération d'assistants de preuve avec une version calculatoire de l'univalence et de les utiliser comme base pour la mise en œuvre de transformations de modèles effectives, afin que le pouvoir de logique interne de l'assistant de preuve puisse être changé dynamiquement—en fonction de la tension entre efficacité et expressivité logique.

Le résultat donnera non seulement un élan déterminant dans la manière dont les chercheurs, aussi bien les informaticiens que les mathématiciens, appréhendent les assistants de preuve, mais fournira aussi l'opportunité de construire de manière effective une nouvelle théorie de types qui formera les nouveaux fondements pour la formalisation des mathématiques. Ce projet sera basé sur l'assistant de preuve Coq car il est dans une dynamique de développement positive permanente, et commence à être utilisé de manière massive dans la communauté scientifique. Mais les résultats du projet pourront tout aussi bien être utilisé dans le cadre d'autres assistants comme Agda ou Lean.

Exemple : la formalisation des monades. Afin d'illustrer la possibilité qu'offrira l'aboutissement de notre projet, considérons le cas d'un informaticien souhaitant formaliser les monades dans les langages de programmations. Les monades constituent une façon très pratique de représenter les effets dans des cadres fonctionnels purs, comme en particulier dans Haskell. Une monade consiste en un constructeur de type `m` qui représente le type de calculs avec effets et deux opérations de plongement des valeurs dans les calculs et de composition de calculs. Ces opérations doivent satisfaire différentes propriétés pour que la notion de calcul induite soit correcte. En Haskell, ces propriétés sont implicitement supposées, et c'est la responsabilité du programmeur de les vérifier sur le papier lorsqu'il définit une nouvelle monade. Il est donc important de pouvoir réimplémenter les bibliothèques Haskell sur les monades en Coq, avec les propriétés vérifiées formellement, et de les extraire ensuite vers Haskell pour obtenir des bibliothèques certifiées qui calculent. Concrètement, définir une monade revient à donner en Coq une instance de la classe de type suivante:

```
Class Monad (m : Type → Type) := {
  unit : ∀ a,   a → m a ;
  bind : ∀ a b, m a → (a → m b) → m b;
  left_identity : ∀ a b x (f : a → m b), bind (unit x) f = f x
  (* plus two other laws *)}.
```

Cela signifie que pour déclarer une nouvelle monade, il est nécessaire de définir les opérations calculatoires en même temps que de prouver leurs propriétés. Regardons plus précisément ce que cela signifie pour une monade bien connue; la monade d'état qui permet aux fonctions de manipuler un état durant leur calcul, comme on peut le faire en programmation impérative. Il est possible de montrer que la monade d'état est une instance de la classe de type `Monad` :

```
Definition State s a := s → s × a.

Instance StateMonad s : Monad (State s) := {
  unit a   := fun a s ⇒ (s,a);
  bind a b := fun c f s ⇒ let (s',a) := c s in f a s';
```

```
  left_identity a b := fun x f ⇒ fun_ext (fun s ⇒ refl)}.
```

L'instance pour la loi d'identité à gauche (comme pour les deux autres lois que nous n'indiquons pas) requiert l'utilisation du principe d'extensionnalité des fonctions. Ainsi, pour extraire complètement le code Haskell du développement Coq, le principe d'extensionnalité fonctionnelle ne doit pas être un axiome. Cela se produit aussi lorsqu'on définit d'autres monades courantes, ce qui indique que tout projet souhaitant formaliser des monades a besoin d'une version native du principe d'extensionnalité des fonctions pour avoir du code extrait qui calcule. Supposons maintenant que deux monades d'état ont été composées. En utilisant l'univalence, la composée des deux monades peut être vue comme une simple monade d'état sur le produit des deux états des monades:

```
Lemma stateCompose s s' a : State (s × s') a = State s (State s' a).
```

Finalement, supposons qu'un développeur définisse une nouvelle monade qui nécessite la loi du tiers exclu. Il pourra alors, au lieu de postuler un nouvel axiome dans Coq, spécifier au début de son développement qu'il se place dans une couche logique qui contient la logique classique, et la loi du tiers exclu deviendra accessible à la compilation à travers une transformation de programme.

Défis. Jusqu'à présent, tous les assistants de preuve basés sur une théorie des types décidable utilisent une notion syntaxique d'égalité. De plus, peu de travaux ont été faits pour étendre le pouvoir interne de la logique en utilisant les idées venant de la logique mathématique. Une des raisons pour tout cela est que la notion syntaxique d'égalité est incompatible avec les transformations de modèles utilisées en logique mathématique pour étendre la logique avec de nouveaux principes. Notre projet se propose donc de relever les défis suivants :

D1. Définir une théorie des types avec une notion native d'univalence

Alors que le principe d'univalence devient communément accepté comme une solution très prometteuse pour donner de nouvelles fondations à la théorie des types et aux mathématiques, il n'a toujours pas été ajouté à un assistant de preuve. C'est en partie car les structures mathématiques (autour de la notion d'∞-groupoïdes) supportant la théorie sont encore sujettes à des discussions actives dans la communauté, et la définition d'une procédure complète et décidable du typage pour la théorie entière pose à la fois des problèmes de complexité calculatoire et de cohérence logique. Une toute première approximation de la théorie homotopique des types a été implémentée dans Epigram sous le nom de "théorie des types observationnelle" [AMS07], mais uniquement pour traiter du principe d'extensionnalité des fonctions et non pour l'univalence. Plus récemment, l'équipe de Thierry Coquand a travaillé sur un modèle calculatoire basé sur modèle des ensembles cubiques [BCH13, CCHM16], mais une formalisation complète en théorie des types intentionnelle est encore un problème ouvert. Mark Bickford[6] a récemment annoncé la formalisation de la théorie cubique des types en NuPRL (basé sur une théorie des types extensionnelle). La formalisation est assez complète mais le principe d'univalence n'a pas été dérivé pour le moment de la sémantique Nuprl.

Le premier défi de notre projet est donc de permettre cette internalisation complète de l'univalence en théorie des types intentionnelle.

D2. Fournir une nouvelle génération d'assistants de preuves avec univalence, sans surcoût calculatoire

[6] `http://www.nuprl.org/wip/Mathematics/cubical!type!theory/index.html`

Une fois qu'une théorie des types avec une version calculatoire de l'univalence aura été mise en place, le prochain défi sera de l'intégrer dans une nouvelle version de Coq, tout en conservant la compatibilité avec les versions précédentes, en particulier au niveau de la vitesse de calcul. En effet, le surcoût de complexité induit par la théorie homotopique des types ne doit pas induire d'explosion de la complexité de la procédure de vérification du typage utilisé si l'on veut que ce nouveau cadre soit accepté rapidement dans la communauté. Concrètement, il faudra s'assurer que le temps de compilation de la librairie standard de Coq reste du même ordre de grandeur entre la nouvelle et l'ancienne version.

D3. Définir une notion de type inductif avec une notion intégrée d'égalité

Durant un groupe de travail au Mathematical Research Institute d'Oberwolfach en 2011, il a été noté que le nouveau point de vue sur l'égalité donné par la théorie homotopique des types permet de généraliser la notion de types inductifs. Les types inductifs sont au cœur de la plupart des formalisations réalisées en Coq car ils forment la brique de base pour définir des structures de données complexes. Ils sont comparables à la notion de types de données abstraits généralisés (GADT) qui peut être trouvée dans Haskell ou OCaml. Un type inductif est défini par un ensemble de constructeurs qui permet de spécifier comment produire des éléments de ce type inductif (comme dans l'exemple des listes ci-dessus). Un des résultats importants de l'année de travail à l'IAS a été de définir une notion de type inductif dans laquelle les constructeurs ne spécifient pas seulement les éléments du type, mais aussi les égalités entre ces éléments. Cette nouvelle construction a été appelée types inductifs supérieurs (higher inductive types, HIT [Uni13]). Intégrer les HIT est très important dans notre projet car ils permettent de définir de nombreuses structures mathématiques comme les types quotients ou d'autres notions de colimites. Cette structure est nécessaire pour le développement du prochain défi (D4). Malheureusement, la définition précise des HIT n'a pas encore été complètement développée. Notre projet se propose donc de fournir la première implémentation des HIT dans Coq.

D4. Étendre modulairement le pouvoir logique de la théorie des types sans axiome

Étendre le pouvoir de la logique en utilisant des transformations de modèles (*e.g.*, la transformation de forcing [JTS12, JLP$^+$16] ou la construction de faisceaux) est un vieux sujet de la logique mathématique. De manière surprenante, ces idées n'ont pas été très regardées dans le cadre de la théorie des types pour étendre modulairement le pouvoir logique d'un assistant de preuve. La raison pour cela est assez simple : avec une notion syntaxique de l'égalité, la structure sous-jacente de la théorie des types n'est pas conforme avec la structure de topos utilisée en logique mathématique; ce qui explique pourquoi une adaptation directe ne marche pas. Cependant, avec une notion univalente de l'égalité, la structure sous-jacente de la théorie des types devient plus proche car elle correspond à la notion d'∞-topos récemment étudiée par Jacob Lurie [Lur09]. Le dernier défi de notre projet pour ces cinq prochaines années est de reformuler et d'implémenter en théorie homotopique des types les transformations de modèles bien connues en logique mathématique, en nous inspirant du travail sur les ∞-topoï.

La figure 1 présente les extensions logiques qui deviendraient possibles avec cette approche : la loi du tiers exclu, l'axiome du choix, un principe d'induction générale ou la logique modale.

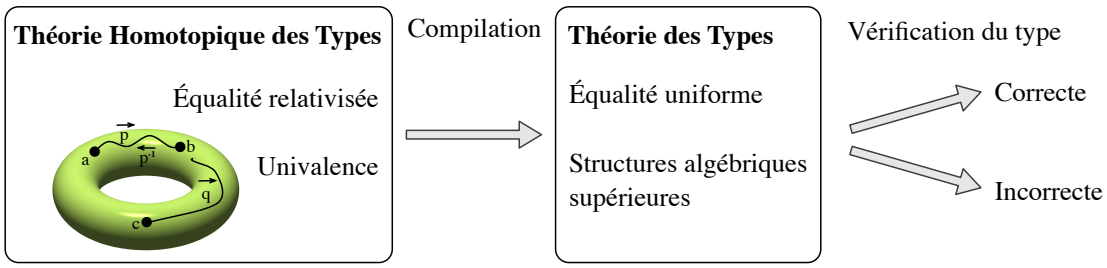


Figure 2: Vérifier le typage en théorie homotopique des types via une phase de compilation

Méthodologie

Après avoir expliqué notre approche globale pour résoudre les défis mentionnés plus haut en utilisant des phases de compilation de théories de types riches vers des théories plus primitives dans Coq, nous détaillons dans cette section les cinq objectifs qui guideront notre projet.

Étendre la théorie des types à travers différentes phases de compilation. Les fondements de notre projet résident dans la mise au point d'une nouvelle génération d'assistants de preuve en rendant possible l'extension de leur logique interne par différentes phases de compilation. En effet, de nos jours, les compilateurs sont formés par composition de différentes phases de compilation, afin de réduire la complexité de chacune de ses phases. Ainsi, il paraît naturel de bénéficier de la même idée pour la définition d'un assistant de preuve complexe. C'est par exemple déjà le cas pour la définition des classes de type de Coq qui sont moralement obtenues à travers une traduction vers un noyau de Coq sans classe de type. Cette approche par phases de compilation s'est déjà montrée fructueuse pour ajouter des points fixes généraux dans Coq en utilisant la traduction de forcing[JTS12, JLP+16]. La première phase de compilation (Fig. 2) correspond à la réalisation des défis D1 et D2. Elle consiste en la définition de l'interprétation de la théorie homotopique des types vers la théorie des types traditionnelle en utilisant une internalisation des ∞-groupoïdes. Je pense sincèrement que l'utilisation de différentes phases de compilation pour modulariser la difficulté de trouver un algorithme de vérification du typage qui soit décidable pour des théories de types complexes est une ligne de travail prometteuse. La seconde phase de compilation (Défi D3) correspond à l'interprétation des types inductifs supérieurs comme des types inductifs de base avec une notion spécifique d'égalité. De cette façon, nous donnerons en même temps un contenu calculatoire aux HIT, et obtiendrons une preuve de équi-consistance avec la théorie des types sans HIT. La troisième phase (Défi D4, Fig. 1) correspond à l'internalisation en théorie homotopique des types des transformations de modèles définies sur les ∞-topoï, comme la construction de préfaisceaux ou de faisceaux. Le succès de nos contributions théoriques ne pourra être garanti qu'à travers le développement d'outils pratiques associés. J'ai donc prévu pour les prochaines années de mettre un effort particulier dans l'implémentation d'une extension de Coq avec le principe d'univalence. Notre but à long terme est d'avoir une extension compatible avec et aussi efficace que la version actuelle de Coq, de manière à ce que cette extension puisse servir de nouvelle version officielle. Notre projet focalise sur l'assistant de preuve Coq car il supporte déjà les classes de types et la gestion implicite d'univers polymorphes, mais les résultats du projet pour aussi bien être déclinés vers d'autres assistants comme Agda ou Lean. Enfin, notre but n'est pas seulement de fournir une nouvelle version de Coq avec univalence mais aussi une intégration des extensions

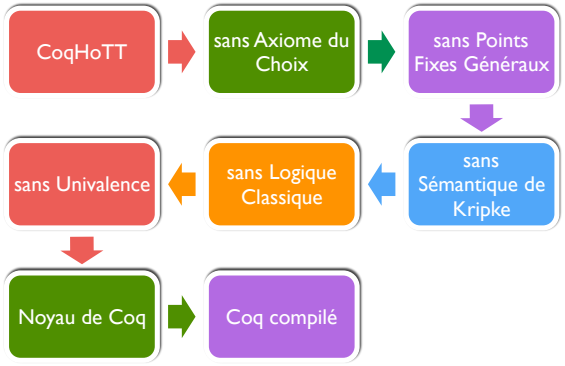


Figure 3: Diverses phases de compilation pour augmenter le pouvoir logique et calculatoire de Coq.

obtenues par l'internalisation des transformations de modèles.

Nous allons construire notre projet autour des cinq objectifs suivants. Le projet global est résumé dans la figure 3.

Objectif 1 : Définir une internalisation de la théorie homotopique des types en Coq

Comme en témoigne les résultats de l'année de travail de scientifiques éminents du domaine (durant l'année "The Univalent Foundations Program" à l'IAS de Princeton [Uni13]), définir un traitement direct et complet de la théorie homopotique des types est une tâche très complexe qui est encore loin d'être acccomplie. Ceci est principalement dû au fait que cela soulève beaucoup de problèmes de cohérence venant de l'utilisation d'une infinité de dimension. Nous pensons que définir une interprétation de la théorie homotopique des types à l'intérieur de la théorie des types elle-même constitue la clé pour une définition complète. Le principal défi ici sera d'internaliser la définition des structures infinies utilisées pour décrire la théorie homotopique des types. Il y a eu plusieurs travaux récents sur le sujet qui servira de base à notre investigation. Le modèle initial de Vladimir Voevodsky est basé sur les ensembles simpliciaux de Kan mais souffre de problèmes de constructivité (comme montré récemment dans [BCP15, BC15]) qui ne le rend pas propice à une mécanisation en Coq. Ces structures infinies pourraient aussi être décrites en utilisant une approche opéradique pour les ∞-groupoïdes. En effet, bien que la construction usuelle des n-catégories strictes via des enrichissements successifs ne marche pas dans le cadre faible, Todd Trimble [Tri99] a proposé récemment d'utiliser certaines opérades pour paramétrer la composition faiblement associative dans les dimensions supérieures afin d'obtenir une définition inductive des n-catégories faibles. Eugenia Cheng et Tom Leinster [CL12] ont proposé récemment une construction co-inductive des ∞-catégories faibles basée sur l'approche de Trimble et la notion de cogèbre finale. Une autre ligne de travaux, initiée par l'équipe de Thierry Coquand, se tourne vers l'utilisation d'un modèle entièrement calculatoire pour les ∞-groupoïdes basé sur les ensembles cubiques [BCH13] donnant lieu à la théorie des types cubiques [CCHM16] pour laquelle un résultat de canonicité a récemment était prouvé pour Simon Huber [Hub16]. Enfin, Thorsten Altenkirch a proposé de construire des ∞-groupoïdes avec une approche plus syntaxique [AR12], en générant d'abord la structure infinie sans ses cohérences, puis en les récupérant en utilisant une approche à la "tous les diagrammes commutent". Il n'est pas encore clair quelle approche sera la plus adaptée

pour notre but, et la détecter constituera déjà une réelle avancée.

Le travail lié à cette tâche ne consistera pas seulement à un développement théorique mais sera solidement ancré sur un socle d'implémentation. En effet, un bon candidat pour la représentation des ∞-groupoïdes ne pourra être détecté qu'à travers la mise en œuvre de sa formalisation en Coq, aussi bien en terme de complexité des définitions mais aussi de preuves associées. De plus, dans notre travail pour internaliser les (wild) ∞-groupoïdes dans Coq [HHT15], il est apparu clairement que ce processus d'internalisation ne fonctionne pour les structures infinies qu'au prix d'une forte automatisation des preuves associées. En particulier, nous aurons besoin d'améliorer et généraliser les techniques de réécriture mises en place par Matthieu Sozeau dans la bibliothèque Setoid. Améliorer cette biliothèque permettra de faire la plupart des preuves grâce à de la réécriture automatique.

Objectif 2: Développer une procédure décidable pour la vérification de type en théorie homotopique des types

Développer une procédure décidable pour la vérification de type en théorie homotopique des types est un des problèmes ouverts les plus importants du sujet. En effet, comme les définitions des objets mathématiques sous-tendant la théorie sont encore sujettes à débat actif, l'établissement d'une procédure de vérification du typage correcte et décidable pose encore de nombreux problèmes. Nous proposons de nous reposer sur le travail qui aura été accompli après avoir atteint notre premier objectif sur l'internalisation d'une interprétation d'un modèle de la théorie homotopique des types afin de définir un algorithme se basant sur une phase de compilation vers la théorie des types de base (Fig. 2). En effet, de nos jours, les compilateurs sont composés en une série d'étapes afin de maîtriser la complexité de manière modulaire. Il semble donc naturel de bénéficier de ces techniques dans la définition d'assistants de preuve complexes. Ceci est déjà le cas pour la définition des classes de type de Coq, qui sont données par une compilation directement dans la théorie sans classe de type.

La théorie requise pour cette axe sera entièrement basée sur l'objectif O1. Le travail restant se situera dans la simplification et l'optimisation de la traduction pour obtenir des termes compilés dont la vérification du typage ne prenne pas beaucoup plus de temps que pour la vérification traditionnelle. Cela peut être atteint par exemple en exhibant des motifs dans la traduction permettant de les pré-typer afin d'accélérer la vérification du typage.

Objectif 3: Implémenter une extension de Coq pour la théorie homotopique des types

L'accomplissement des deux premiers défis nous ouvrira la voie à l'implémentation complète d'une extension de Coq pour la théorie homotopique des types. À l'inverse des deux premiers défis, cette partie de notre projet n'est pas ambitieuse du point de vue théorique, mais plutôt au niveau de l'ingénierie logicielle. Elle requiert l'implémentation de la phase de compilation d'une transformation de programmes complexe tout en conservant des performances proches de la version de Coq n'utilisant pas cette transformation. En effet, la complexité supplémentaire de la théorie homotopique des types ne doit pas induire une explosion combinatoire si on veut que la nouvelle version devienne le standard. Nous pensons aussi intégrer des idées venant de la programmation par aspect afin d'étendre le langage de tactique de Coq avec des constructions permettant une utilisation plus facile de l'univalence en informatique. En effet, la notion de points de coupe (qui détecte les points d'intérêt lors de

l'exécution d'un programme) peut être vue comme un moyen de décrire les endroits où utiliser des isomorphismes, et un advice (qui indique quel programme exécuter) peut être vu comme l'isomorphisme à appliquer en ce point. Ainsi, il sera plus facile de changer à la volée la présentation des données à l'exécution.

Cette tâche est principalement technique. Notre expérience avec l'implémentation de la traduction de forcing [JTS12, JLP+16] nous montre qu'il sera sûrement nécessaire d'améliorer la vérification du typage de Coq pour obtenir un résultat satisfaisant. En effet, la vérification est basée sur de nombreuses heuristiques qui peuvent être améliorées en ayant une meilleure compréhension de la forme des termes qui seront obtenus après compilation.

Objectif 4: Définir et implémenter une notion générale de types inductifs supérieurs

Après avoir fourni un cadre pour l'intégration du principe d'univalence dans Coq, l'autre point important de la théorie homotopique des types à intégrer sera la notion de types inductifs supérieurs (Fig. 2, gauche). Ces types sont au cœur de la formalisation de la théorie de l'homotopie dans un assistant de preuve. Leur donner un contenu calculatoire permettra aux mathématiciens, non seulement de formaliser et prouver des théorèmes en théorie de l'homotopie, mais aussi d'utiliser la théorie des types pour calculer les groupes d'homotopie d'objets complexes. En effet, les théoriciens de l'homotopie sont maintenant confrontés à une explosion de la complexité des objets qu'ils manipulent dans leurs travaux sur la classification des espaces topologiques. il est maintenant de notoriété publique que cette explosion nécessite l'utilisation d'ordinateurs pour assister les mathématiciens.

La notion de types inductifs supérieurs a aussi des conséquences en informatique car elle fournit un nouveau constructeur de types qui vient avec sa propre notion d'égalité. Par exemple, les systèmes de contrôle de version peuvent être modélisé avec des types inductifs supérieurs dans lesquels les actions "commit" et "revert" correspondent aux applications des égalités venant avec ce HIT [AMLH14].

Au sein de notre projet, l'intégration des types inductifs supérieurs sera aussi très important pour le développement du dernier objectif.

Objectif 5: Étendre la théorie homotopique des types avec de nouveaux principes logiques et calculatoires

Avec une implémentation pratique et effective de la théorie homotopique des types, nous avons pour but de réutiliser les transformations de modèles bien connues en logique mathématique afin de pouvoir augmenter le pouvoir logique de la théorie des types avec de nouveaux principes. Le défi ici est de donner un sens, à travers différentes phases de compilation, à des axiomes souvent utilisés en mathématique, comme la loi du tiers exclu ou l'axiome du choix (Fig. 1). Comme ajouter ces nouveaux principes peut avoir un impact sur les temps de calcul et peut parasiter le mécanisme d'extraction, notre but est de pouvoir étendre de manière modulaire la logique interne de Coq. Nous avons déjà montré [JTS12, JLP+16] que le forcing (qui correspond à la construction de préfaisceaux) permet de sublimer la logique de Coq avec des nouveaux constructeurs, comme par exemple des types inductifs non restreints (à ne pas confondre avec les types inductifs supérieurs). Notre projet est d'appliquer cette technique à d'autres constructions qui nécessitent l'univalence pour être correctes, comme par exemple la construction de faisceaux. Cela permettra d'implémenter dans la théorie des types la fameuse traduction de Gödel de la logique

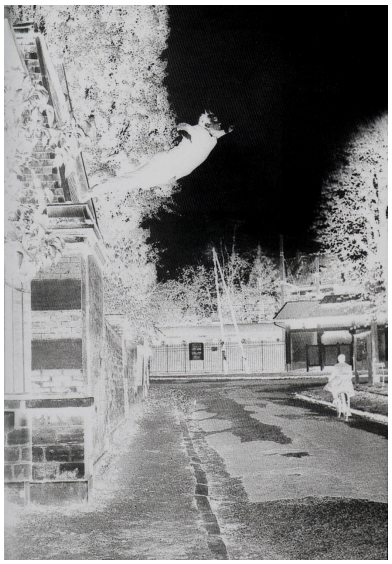
classique vers la logique intuitioniste, donnant ainsi un sens calculatoire au tiers exclu.

La structure sous-jacente de la théorie homotopique des types est celle des ∞-topoï, récemment étudiés par Jacob Lurie [Lur09]. Le principal défi à notre niveau sera de rendre concrètes et effectives les transformations décrites dans le cadre des ∞-topoï. En effet, même si les préfaisceaux et les faisceaux ont déjà été définis dans ce cadre, leurs définitions sont très abstraites et nécessitent un gros travail d'adaptation pour les rendre suffisamment effectives pour être implémenter en théorie homotopique des types. Par exemple, une présentation effective des faisceaux nécessite de reformuler les faisceaux dans les ∞-topoï en utilisant la notion de topologie de Lawvere-Tierney [MM92] qui n'a pour le moment jamais été définie dans ce cadre.

Pour cette tâche, nous souhaitons reprendre nos travaux sur le forcing et implémenter une internalisation de la notion de faisceaux à la Lawvere-Tierney dans Coq. En utilisant ces traductions, il deviendra possible d'ajouter calculatoirement (*i.e.*, sans axiome) de nouveaux principes à la théorie de Coq. Comme nous anticipons des allongements significatifs des temps d'exécution pour l'utilisation de ces nouveaux principes, nous aurons à trouver des mécanismes pour bien séparer la compilation des modules qui utilisent ces nouveaux principes des autres modules. Ainsi, le surcoût calculatoire ne sera payé que dans les modules qui font un usage effectif de ces principes, ce qui rendra notre approche beaucoup plus utilisable en pratique.

Chapter 1

Looking Back into the Future



Le saut dans le vide (négatif),
Klein (1960)

Contents

1.1 A Retrospective Introspection

Remark 2

Although the rest of the manuscript is written in a more conventional way, this section uses the first-person perspective to stress that it develops a personal point of view.

Semantics of compilation. During an internship at Microsoft Research Cambridge in 2007, I started to study the semantics of compilation [BT09] under the delightful supervision of Nick Benton. My subject of interest at this time was mainly on denotational semantics of programming languages, but I started to realize[1] that compiling—or rather translating—a program written in a complex language into a program written in a simpler language was a very primitive and efficient way of giving a meaning to a complex language.

[1] I also realized that I like the Coq proof assistant [CDT15] a lot, which should be obvious from the reading of the rest of this manuscript.

Aspect Oriented Programming. This remark pursued me as I started to study the semantics of Aspect Oriented Programming (AOP) when I entered the Ascola team (Aspects Components Languages) in 2009. AOP [KLM+97] is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. These are concerns that cannot be defined using traditional modularity mechanisms, such as classes, modules, or components. Tracing and security are two examples of crosscutting concerns, and thus are subject to scattering and tangling. The idea of AOP is to bring back all the tracing code together within a new kind of module, called an aspect. The two main constituents of an aspect are pointcuts and pieces of advice. A pointcut selects the execution points of interest in the base program, for instance, method calls in the case of tracing. A piece of advice defines what to do when an execution point is selected, typically display a message before and after a call in the case of tracing. The process of composing aspects with a base program, recreating the program which would have been implemented without AOP, is called *weaving*.

My first work on the meaning of a distributive aspect calculus [Tab10] has been done by translating it into the more primitive join calculus [FG96a], thus considering the weaving process as a program transformation. I have then studied the semantics of functional AOP [Tab11, Tab12] in the light of monadic embeddings, which constitute a very famous program transformation in functional programming. Monads [Mog91] are a mechanism to embed and reason about computational effects such as state, I/O, or exception-handling in purely functional languages like Haskell. Defining the semantics of functional AOP using a monad allows to be modular on the different possible semantics of aspects (*e.g.*, aspect deployment, interference), and allows for a direct implementation in functional languages such as Haskell [TFT13, FTT14b, FSTT14].

By using ideas coming from the scoping of aspects, I have developed with Ismael Figueroa and Éric Tanter a monadic definition in Haskell of effect capabilities [FTT14a, FTT15] which allows to impose certain security policy on the effects that are used in an Haskell program. Together with the monadic setting, it uses heavily the type class mechanism of Haskell to decide the level of permission using an implementation of permission lattices as type classes.

Forcing in Type Theory. In the meantime, I was using more and more Coq to formalize (part of) the proof I was doing on paper. I realized then that even if type theory (or more precisely the Calculus of Inductive Construction, the theory behind Coq) was very powerful, we were lacking the possibility to enhance the power of the logic or computational behaviour easily.

This has lead me to develop with Guilhem Jaber and Matthieu Sozeau a version of forcing in type theory [JTS12]. Forcing is a method originally designed by Paul Cohen to prove the independence of the Continuum Hypothesis from the axiomatic set theory ZFC [Coh66]. The main idea is to add new objects which can be *approximate* in the ground system, using the so called *forcing conditions*. To adapt these ideas to type theory, rather than using the usual set theoretic approach of Forcing, we have used its restatement in terms of Category Theory that Bill Lawvere and Myles Tierney [Tie72] have pursued using Sheaves Topos. Recently, after the works of Jean-Louis Krivine [Kri11] and Alexandre Miquel [Miq11a], it became accepted that forcing techniques are of great interest for the extension of the Curry-Howard correspondence. The starting point of our work was to connect these two observations:

> "Intuitionistic forcing for type theory is an internalization
> of the presheaf construction in type theory"

The idea is to extend a base type theory—called the *ground system*—with new principles, getting a new type theory—called the *forcing layer*. Terms and types of the forcing layer can be translated to the ground system using the *forcing translation*. Then, the forcing layer may satisfy new logical or computational principles, *e.g.*, the existence of general fixpoints when forcing with natural numbers as forcing conditions—which corresponds to considering the topos of trees.

More recently, we have realized that this forcing program transformation was lacking some computational properties that make it not really usable in practice because of coherence issues. This is why we have shifted from a call-by-value to a call-by-name interpretation, under the initiative of Pierre-Marie Pédrot, leading to a new version of the forcing translation [JLP+16] directly implementable as a Coq plugin. Indeed, our translation is directly implementable because it interprets conversion—*i.e.*, the fact that two terms are convertible using definitional equality—as conversion. On the contrary, in many other works and in particular when dealing with models of type theory, conversion is interpreted as a propositional equality or as an external notion of equality provided by the model. In that case, we are facing a coherence issue, because conversion must then be interpreted as an explicit transportation of one term into the other and can not be implicit anymore. Although this coherence issue has been quite widely studied, in particular in the recent work of Pierre-Louis Curien, Richard Garner and Martin Hofmann [CGH14], no solution is available to produce a correct program transformation of type theory from such models/transformations. This observation leads me to the following slogan, which constitutes one of the cornerstone of the research direction developed in Section 1.2:

> "A correct and implementable program transformations
> must preserve type conversion on the nose."

More precisely, I believe that it should be possible to modify already existing transformations to preserve conversion as we have done for forcing by going from a call-by-value to a call-by-name interpretation of forcing.

Dependent Interoperability. Another reflection that came to me when discussing with Éric Tanter about gradual typing in programming languages, is that Coq were lacking similar mechanism to allow a smoother learning time for developers aiming at moving from a simply typed to a dependently typed setting.[2] Actually, we realized that the problem also occurs in the other direction, when transforming a program written in Coq to a program written in a simply typed language such as OCaml or Haskell—a process known as *program extraction*. In that case, the absence of graduality leads to a great loose of properties, from the original code to the extracted code. In that setting, I have worked with Pierre-Évariste Dagand and Éric Tanter on dependent interoperability [TT15, DTT16] between Coq and OCaml/Haskell which can be seen as a way to automatically improve program extraction, *i.e.*, the program transformation from Coq to a simply typed language.

1.1.1 Overview of the Rest of the Document

The next section presents my research program for the five coming years, and maybe more.[3] After that, the rest of the document is devoted to the presentation of three different works on quite different topics but sharing the common idea of using program

[2] We use the term "simply typed" to mean "non-dependently typed", *i.e.*, we do not rule out parametric polymorphism.

[3] Most of it will be developed inside the *CoqHoTT* project—ERC starting grant 637339.

transformation to handle logical or computational complexity. All those works have been published and are here gathered, standardized, and revisited to make the manuscript more self-contained.

Chapter 2 describes a transformation from Coq programs to Coq programs, in order to increase the logical and computational power of Coq. This chapter directly comes from the paper *The definitional side of the Forcing* [JLP+16] which introduces a call-by-name version of forcing in Coq. This is joint work with Guihlem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot and Matthieu Sozeau.

Chapter 3 presents a distributed aspect calculus based on the join calculus. The complexity of aspects deployment and weaving in the context of distributed processes is handled by giving a direct translation from the aspect join calculus to the core join calculus, giving at the same time a semantics and an implementation of the aspect join calculus. This is a revisited and extended version, done with Éric Tanter, of the paper *A theory of distributed aspects* [Tab10].

Chapter 4 describes, to a certain extent, a program transformation from a Coq program with dependent types, to a Haskell or OCaml program by moving some static checks into dynamic ones. This chapter directly comes from the paper *Partial Type Equivalences for Verified Dependent Interoperability* [DTT16]. This is joint work with Pierre-Évariste Dagand and Éric Tanter.

1.2 Future Line of Reasearch

Equality in proof assistants based on type theory. Every year, software bugs cost hundreds of millions of euros to companies and administrations, as assessed by Intel Pentium division bug or Ariane 5 first flight failure. Hence, software quality is a notion that becomes more and more prevalent, going beyond the usual scope of embedded systems. In particular, the development of tools to construct softwares that respect a given specification is a major challenge of current and future researches in computer science. Interactive theorem provers based on type theory, such as Coq [CDT15], equipped with an extraction mechanism that produces a certified program from a proof, are currently gaining traction towards this direction. Indeed, they have shown their efficiency to prove correctness of important pieces of software like the C compiler of the CompCert project [Ler06]. One main interest of using such theorem provers from a computer science point of view is the ability to extract the code that has been proven directly from the proof, being able to run it as any other pieces of code. For instance, the CompCert compiler extracted from its certification is reliable and efficient, running only 15% slower than GCC 4 at optimization level 2 (i.e., ggc -O2), a level of optimization that was considered before as highly unreliable.

Unfortunately, the democratization of such interactive theorem provers suffers from a major drawback, the mismatch between the conception of equality in mathematics and equality in type theory. Indeed, some basic principles that are used implicitly in mathematics—such as Church principle of proposition extensionality, which says that two formulas are equal when they are logically equivalent—are not derivable in (Martin-Löf) type theory. More problematically from a computer science point of view, the basic concept of two functions being equal when they are equal at every "point" of their domain is also not derivable and need to be set as an axiom:

```
Axiom fun_ext : ∀ A B (f g : A → B), (∀ x, f x = g x) → f = g.
```

Of course, those principles are consistent with type theory and adding them as axioms is safe. But any development using them in a definition will produce a piece of code that

does not compute, being stuck at points where axioms have been used, because axioms are computational black boxes. To understand this mismatch, we need to take a deeper look at the notion of equality in type theory. It is defined using identity types Id_A with only one introduction rule (uniform with respect to A):

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathrm{refl}_A\ x : \mathrm{Id}_A\ x\ x}$$

This means that the only way to prove that two terms t and u are equal (or more precisely to inhabit $\mathrm{Id}_A\ t\ u$ without additional assumptions) is by reflexivity, which means that the two terms must be convertible—in the sense that they have the same normal form modulo β-reduction (and some other reduction rules). To some extend, this notion of equality is eminently syntactic. The reason why identity types are used in type theory is because they come with a nice elimination principle which allows to substitute a term by an Id-equivalent term anywhere in a type. This is Leibniz's principle of indiscernability of identicals.

To the opposite, the notion of equality in mathematics is eminently semantics, with a definition that is relative to the type of terms being considered. This is why the functional extensionality principle is easy to set in this setting, because the definition of equality is specialized accordingly when it comes to function types. The problem with that point of view is that it does not come with a generic substitution principle and thus is difficult to use it in a programming language.

This mismatch prevents proof developments from using more advanced semantical tools in their reasoning because of the explicit manipulation of mathematical equality together with axioms. Of course, some important work has been done to work around this difficulty by defining other notions of equalities beside, using the well known notion of setoids [BCP03]. But this does not provide equality as we mean it on black board because we can not use the generic substitution principle on equality available for identity types. If a working computer scientist wants to use a setoid-based notion of equality and do some rewriting with it in an expression, she needs to prove beforehand that this expression validates the substitution principle. And when the considered expression comes from a separated development, this can be a very complex task, even for a single expression. This non-uniform treatment of equality causes problems when proofs need to be combined, which is one of the key stones of mathematical developments. As a consequence, significant Coq developments have been done, but only by virtuosos playing around with advanced concepts of both computer science and mathematics, which excludes by definition a working computer scientist (and also working mathematician). This project aims at solving this issue by delivering a proof assistant in which a working computer scientist or mathematician can manipulate the notion of equality as she does on black board.

The birth of homotopy type theory. To correct the fundamental mismatch between equality in mathematics and in type theory, Hofmann and Streicher have introduced a new point of view on type theory, where types are not viewed as simple sets but as sets enriched with an homotopical structure [HS96]. This way, each type comes with a specialized notion of equality and the homotopical structure describes how to substitute a term by another equivalent one in a type. Voevodsky recognized recently that this simplicial interpretation of type theory satisfies a further crucial property, dubbed univalence, which had not previously been considered in type theory. The univalence principle coarsely says that two types (or structures) `T` and `U` are equal if and only if they are equivalent (or

isomorphic)[4]:

```
Definition univalence := ∀ T U : Type, (T = U) ≅ (T ≅ U).
```

The univalence principle subsumes the other missing principles of equality and is the key to equip type theory with a notion of equality that is compatible with traditional mathematical reasonings. In recent developments of HoTT [Uni13], the univalence principle is added to the type theory of Coq (or Agda) in the form of a new axiom. This has far-reaching consequences in the formalization of homotopy theory, such as the proof that the first homotopy group of the sphere is equivalent to $\mathbb{Z}$ [LS13]. But, the univalence principle has also a very interesting interpretation in computer science. It allows to use a function/method of a library—that has been defined on a data type `A`—on another data type `B`, as soon as there is an explicit type isomorphism between `A` and `B`. Take for instance `A` to be the inductive type for lists.

```
Inductive list₁ A : Type := nil₁ : list₁ A | cons₁ : A → list₂ A → list₂ A.
```

Suppose now a developer has used its own inductive type for lists in its development and for convenience has defined the constructor `cons` by taking the arguments in a reverse order:

```
Inductive list₂ A : Type := nil₂ : list₂ A | cons₂ : list₁ A → A → list₁ A.
```

Using univalence, no matter which type representation has been used, it becomes easy to switch from one type to another because those the two types are isomorphic and can be considered as equal. This change of base/type is very frequent and can be needed for different reasons: for efficiency or algorithmic convenience (think about the many different ways of representing a list), for compatibility between data representation (*e.g.*, meter vs inch, US vs european date format)... Using univalence allows at the same time to perform an automatic conversion of a program acting on `A` to a program acting on `B`, and to have the guarantee that both programs can not be distinguished from the outside. This can be very useful for software integration. But HoTT is eminently higher-dimensional. This means that we can also use univalence to replace a program by another program as soon as we can prove that there are isomorphic, *i.e.*, observationally equivalent. This can be seen as a way to perform safe optimization.

1.2.1 Objectives over the five coming years

Unfortunately, stating univalence as an axiom breaks one fundamental property of proofs mechanized in a type theory based proof assistant: it breaks the extraction mechanism, that is the ability to compute with a program that makes use of this axiom. The starting point of our project is to build a system where univalence is a build-in property of the system—not an axiom. Then, the underlying type theory will be more regular, being closed to the structure of a topos (actually, being an ∞-topos), which is a fundamental notion in mathematical logic. This point is very important as it allows to revisit logical model transformation usually performed on topoi to transformation at the level of homotopy type theory. Concretely, it will become possible to extend modularly the power of the logic governing the proof assistant, *e.g.*, by adding a general induction principle, axioms of classical logic, the axiom of choice or modal logic (Fig. 1.1).

[4] Actually, as formally defined in Section 2.6.3, it says more precisely that the canocical map from `T = U` to `T ≅ U` is an equivalence.

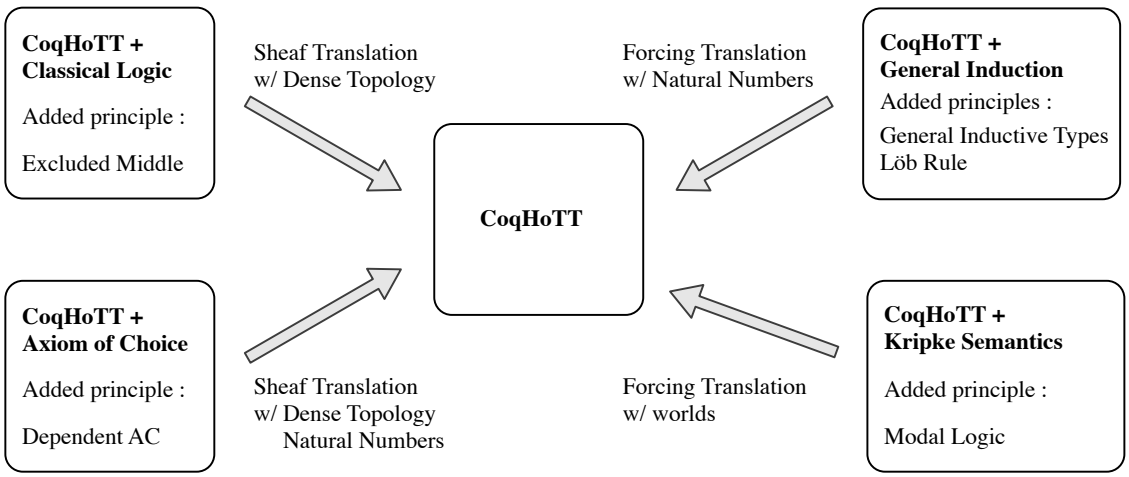


Figure 1.1: Extension of Coq using program transformations of mathematical logic

> The main goal of our project is to provide a new generation of proof assistants with a computational version of univalence and use them as a base to implement effective logical model transformation so that the power of the internal logic of the proof assistant needed to prove the correctness of a program can be decided and changed at compile time—according to a trade-off between efficiency and logical expressivity.

The results would not only represent a huge step forward in the way researchers, both computer scientists and mathematicians, apprehend proof assistants, but also give the opportunity to develop fully a new type theory that will give new foundations to formalized mathematics and computer science. The project will be based on the Coq proof assistant because it is in a real and permanent active development and is now widely used in the scientific community. But the outcomes of the project could also be used to extend other proof assistant based on type theory, for example Agda or Lean.

Example: formalization of monads. To illustrate the possibilities offered by our proposal, we consider the case of a computer scientist that wants to formalize monads in functional programming. Monads constitute a very convenient way to represent effects in a purely functional setting, and particularly in Haskell. A monad consists in a type constructor `m` that represents the type of computation with effects and two operations, unit and bind, that correspond to lifting of values into computations and composition of computations. Those operations must fulfill several properties to allow the correct composition of monadic functions. In Haskell, those properties are implicitly assumed, and it is of the responsibility of the programmer to check them on paper when she defines a new monad. Therefore, it is important to be able to reimplement Haskell monad libraries in Coq, with properties formally checked, and then extract the code back to get certified Haskell monad libraries. Concretely, the data of a monad can be gather into the following Coq type class:

```
Class Monad (m : Type → Type) := {
  unit : ∀ a,  a → m a ;
  bind : ∀ a b, m a → (a → m b) → m b;
  left_identity : ∀ a b x (f : a → m b), bind (unit x) f = f x
  (* plus two other laws *)}.
```

This means that to declare that a type constructor defines a monad, it is necessary to define the computation operations together with their properties. Let us see what it means for a well-known monad; the state monad that allows functions to manipulate a state during the computation, in the same way as it is done in imperative programming. The state monad can be shown to be an instance of the monad type class in the following way:

```
Definition State s a := s → s × a.

Instance StateMonad s : Monad (State s) := {
  unit a   := fun a s ⇒ (s,a);
  bind a b := fun c f s ⇒ let (s',a) := c s in f a s';
  left_identity a b := fun x f ⇒ fun_ext (fun s ⇒ refl)}.
```

The instantiation of the left identity law (as well as the two other laws) requires the use of functional extensionality. Therefore, to extract Haskell code from a Coq development that makes a relevant use of monadic laws, functional extensionality need not be an axiom. This also happens when one tries to define other typical monads, which means that any monad formalization project would require native functional extensionality to provide usable extracted code. Suppose now that two state monads are composed. Using univalence, this composed monad can be seen as a single state monad, because the type of the composed states is isomorphic to the type of their product:

```
Lemma stateCompose s s' a : State (s × s') a = State s (State s' a).
```

Finally, suppose a developer defines a new monad that requires the law of excluded middle to define one of the monad laws. Then, instead of having to set a new axiom, the developer just has to specify at the beginning of its development that she wants to work in a classical setting and the law of excluded middle will be available through a logical model transformation.

Issues. So far, all proof assistants based on a decidable type theory use a syntactic native notion of equality. Also, no work has been done to modularly extend the internal power of the logic using ideas coming from mathematical logic. One of the reason is that the syntactic notion of equality is incompatible with model transformations used in mathematical logic to extend a logic with new principles. Therefore, our project will address the following issues:

I1. Set up a type theory with a build-in notion of univalence

While the univalence principle becomes widely accepted as a very promising solution to give new foundations to mathematics and type theory, it has not yet been added to a proof assistant. This is because the mathematical structures (around the notion of ∞-groupoids) sustaining the theory are still under active discussion, and the establishment of a correct and decidable type checking procedure for the whole theory causes both computational complexity and logical coherence issues. A first-stage approximation to homotopy type theory has been implemented in Epigram as Observational Type Theory [AMS07], but it only deals with functional extensionality and does not capture univalence. A lot of work has been done by Coquand's team on the computational meaning of univalence in the cubical sets model [BCH13, CCHM16] but a complete formalization in intensional type theory is still an open problem. It has to be noticed that Mark Bickford[5] has recently announced a formalization of cubical type theoy in NuPRL (based on an extensional

[5] http://www.nuprl.org/wip/Mathematics/cubical!type!theory/index.html

type theory). The formalization is quite complete although univalence has not been derived for the moment from the Nuprl semantics.

The first challenge of our project is to achieve a complete internalization of univalence in intensional type theory.

I2. Provide a new generation of proof assistants with univalence, without overhead

After a type theory with computational version of univalence has been set up, the next issue will be to integrate it to a new version of Coq, while keeping compatibility with previous versions, in particular from a performance point of view. Indeed, the additional complexity of homotopy type theory should not induce a blow up in the type checking procedure used by the software if we want our new framework to become rapidly accepted in the community. Concretely, we will make sure that the compilation time of Coq 's Standard Library will be of the same order of magnitude.

I3. Define a notion of inductive type with embedded notion of equality

During a workshop at the Mathematical Research Institute of Oberwolfach in 2011, it has been noticed that this new point of view on equality allows to generalize the notion of inductive types. Inductive types are at the heart of formalization in type theory in that they are the basic bricks to build complex data structures. They are similar to the notion of (generalized) abstract data types that can be found in Haskell or OCaml. An inductive type is defined by a set of generators that specified how to produce elements of that type (see the example of lists above). A very important outcomes of the year of work at IAS has been the definition inductive types in which generators do not only specified new elements of the type, but also new equalities between those elements. This new construction has been coined Higher Inductive Types (HIT) in [Uni13]. Integrating HIT is very important for the project because it allows to define many mathematical structures such as quotient types or other notions of colimits. This structure will be mandatory for the development of the next issue (I4). Unfortunately, the precise definition of HITs has not been worked out completely. Our proposal aims at providing a first solution to the implementation of HITs.

I4. Extend modularly the logical power of type theory without axioms

Extending the power of a logic using model transformation (*e.g.*, forcing transformation [JTS12, JLP+16] or the sheaf construction) is an old subject of mathematical logic. Surprisingly, those ideas have not been much investigated in the setting of type theory to extend modularly the logical power of proof assistant. The reason for that is simple: with a syntactic notion of equality, the underlying structure of type theory is not conform to the structure of topos used in mathematical logic, explaining why a direct adaptation was not possible. However, with an univalent notion of equality, the structure of type theory becomes closer as it corresponds to the notion of ∞-topos recently studied by Jacob Lurie [Lur09]. The last issue of our proposal is to reformulate and implement well-known logical model transformations in the language of HoTT, taking inspiration in the work on ∞-topos. Figure 1.1 presents the logical extensions that should be possible with this approach: law of excluded middle, dependent choice, general recursion or modal logic.

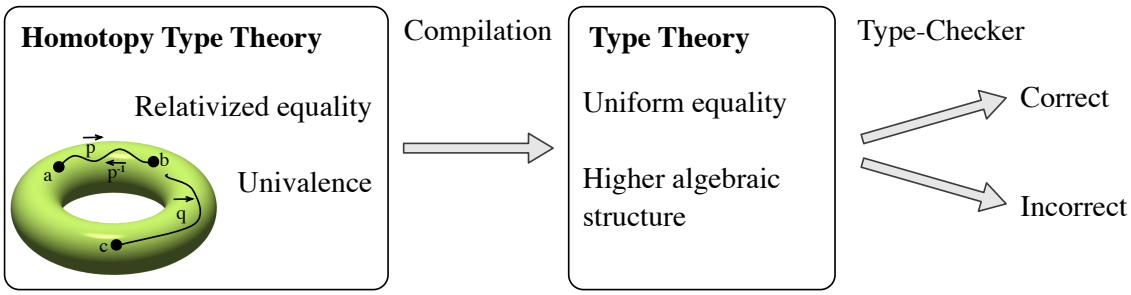


Figure 1.2: Type-checking homotopy type theory via a compilation phase

1.2.2 Methodology

After explaining our general approach to solve above issues using compilation phases from rich type theories to the core type theory of Coq, we detail in this section the five milestones that will guide our research project.

Extending type theory with different compilation phases. The foundation of our project is to design a new generation of proof assistant by extending the logic modularly using distinct compilation phases. Indeed, nowadays, compilers are composed of a series of stages to break the complexity modularly. Therefore, it is natural to benefit from this idea in the definition of complex proof assistants. This is for instance already the case for the definition of type classes in Coq, which are morally given via a translation into a type-class free kernel. This compilation phase approach has already shown useful to add unrestricted inductive types using the forcing transformation [JTS12, JLP+16]. The first compilation phase (Fig. 1.2) corresponds to the realization of issues I1 and I2. It consists in the definition of an interpretation of homotopy type theory into the traditional type theory using an internalization of the ∞-groupoid interpretation. I am confident that a compilation phase into traditional type theory to modularize the difficulty of finding a decidable type checking algorithm for HoTT is a very promising line of work. The second compilation phase (Issue I3) corresponds to the interpretation of higher inductive types as plain inductive type with a specific notion of equality. This way, we will at the same time give computational meaning to HITs and get a proof of consistency with respect to type theory. The third compilation phase (IssueI4, Fig. 1.1) corresponds to the internalization in HoTT of logical model transformations performed on ∞-topoi, such as presheaves or sheaves constructions.

The success of our theoretical contributions can only be assessed in the light of a practical tool. Therefore, I plan to put a strong emphasis on the implementation of an extension of Coq with the univalence principle. Ultimately, our goal is to make this extension compatible with and as efficient as the current version of Coq so that it can serve as the base for a new release of Coq. Our project will be based on the Coq proof assistant because it supports universe polymorphism and type classes but the resulting outcomes of the project could be very well adapted to be used in other proof assistants based on type theory, such as Agda or Epigram. Ultimately, the project should result not only in the release of new version of Coq but also in the integration of logical extensions defined by model transformations.

We plan to develop our framework along the five following objectives, which are described below. The overall project is summarize in Figure 1.3.

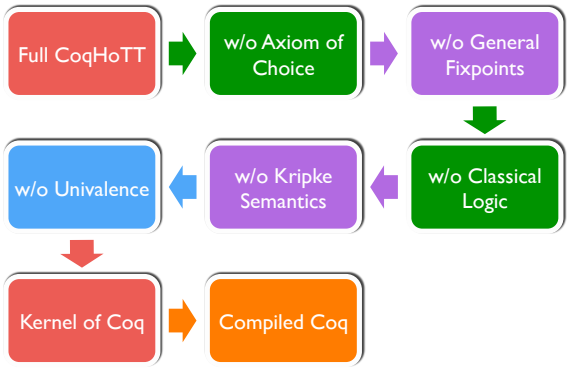


Figure 1.3: Multiple compilation phases to increase the logical and computational power of Coq.

Objective 1: Define an internalization of homotopy type theory in Coq

As witnessed by the outcome of a whole year of work of leading mathematicians and computer scientists of the field (during The Univalent Foundations Program at the Institute for Advanced Study in Princeton [Uni13]), developing a direct complete treatment of homotopy type theory is a very complex task that is far from being accomplished. This is mainly because it raises a lot of coherence issues coming from the use of infinitely many dimensions. We are confident that defining an interpretation of homotopy type theory inside type theory is the key to successfully interpret it. The main challenge of Objective 1 is to internalize the definition of the infinite structure involved in homotopy type theory. There are various recent current works on the subject on which we plan to base our investigations. The original model of Voevodsky is based on Kan simplicial sets but suffers from constructivity issues (as shown recently in [BCP15, BC15]) which does not make it a good candidate for mechanization. The infinite structure could be described using an operadic approach to ∞-groupoids. Indeed, although the usual enriched inductive construction of strict n-categories is known to be broken in the weak setting, Trimble [Tri99] has recently proposed to use specific operads to parametrize the weakly associative composition in the higher-dimensional structures and recover an inductive definition of weak n-categories. Cheng and Leinster [CL12] have recently proposed a co-inductive construction of weak ∞-categories based on Trimble's approach and on the notion of terminal coalgebras. Another line of work, initiated by Coquand's team, aims to look at fully computational notion of ∞-groupoids, for instance by studying cubical sets [BCH13], giving rise to cubical type theory [CCHM16] for which a canonicity result has recently been proven by Simon Huber [Hub16]. Finally, Altenkirch has proposed to construct ∞-groupoids in a more syntactic way [AR12], by generating the structure without higher-dimensional coherences and getting back coherences using an "all diagrams commute" approach. It is not clear what is the best description to be used but this question is currently a highly discussed topic among homotopy type theorists.

This task does not only consist in theoretical investigation but is also strongly based on implementation. Indeed, a good candidate to represent ∞-groupoids can only be discovered by trying to formalize all candidates in Coq and see which one is more suitable for efficiency and proofs. Moreover, in our work to internalize the (wild) groupoid construction in Coq [HHT15], it appears clearly that this internalization

process can not be lifted to infinite dimensions without an efficient automatization of the proofs involved. In particular, we plan to improve and generalize with Matthieu Sozeau actual coq rewriting techniques of the library Setoid. Improving it should allow to perform (part of) many proofs by automatic rewriting.

Objective 2: Develop a decidable type checking procedure for homotopy type theory

Developing a decidable type checking procedure for homotopy type theory is one of the main open issue of the topic. Indeed, as the definition of the mathematical objects sustaining the theory are still under active discussion, the establishment of a correct and decidable type checking procedure for the whole theory causes both computational complexity and logical coherence issues. I propose to rely on the work of Objective 1 on internalization of the interpretation HoTT in type theory to define the algorithm via a compilation phase to type theory (Fig. 1.2).

The theory required by this task will be fully based on Objective 1. The remaining step will be to try to simplify and/or optimized the translation in order to obtain terms of traditional type theory that can be type checked in a reasonable amount of time. This can be achieved for instance by exhibiting patterns in the translation that can be pre-typed in order to use them as basic bricks and accelerate the type-checking procedure.

Objective 3: Implement an extension of Coq to homotopy type theory

The accomplishment of the two previous challenges will give a way to implement an extension of Coq to homotopy type theory. On the contrary to previous challenges, this part of the project is very challenging not from a conceptual but from a software engineering point of view. It requires to implement the compilation phase of the complex transformation of previous objectives while keeping performances closed to the current version of Coq. Indeed, the additional complexity of homotopy type theory should not induce a blow up in the type checking procedure used by the software. We will also integrate ideas coming from Aspect Oriented Programming to extend the language of Coq with constructs that ease the use of univalence in programming. Indeed, the notion of pointcuts (that detect points of interest in a program execution) can be seen as a way to write down basic type isomorphisms and advices (that express the program to be executed when a pointcut matches) can be seen as a way to write down program isomorphisms. Thus, the possibility to change data representation or implementation on the fly will be a native feature of the language.

This task is mainly an implementation task. Our experience with the implementation of the forcing translation [JTS12, JLP+16] has shown that it will be necessary to improve existing Coq type checking procedure to end up with an effective software. Indeed, the type checking is made of a lot of heuristics that can be strongly improved by having a good knowledge of the shape of the term that will be typed. In the case of a compilation phase approach, analyzing this shape to improve heuristics is crucial to get an efficient type checking procedure.

Objective 4: Define and implement a general notion of higher inductive types

After the interpretation of HoTT in Coq has been set up, we will use it to give a meaning to the newly introduced notion of higher inductive types (HIT) (Fig. 1.2, left). Those objects are at the heart of the formalization of homotopy theory inside a proof assistant. Giving them a computational meaning will allow mathematicians

not only to formalize and prove theorems in homotopy theory, but also to use type theory to compute the homotopy groups of complex objects. Indeed, homotopy theorists now face a computational blow up in their work on the classification of topological spaces. This blow up is commonly believed to require the use of a computer to assist mathematicians. But the notion of HIT as also consequences in computer science as it provides a new type former that provides constructors together with (relevant) equalities. For instance, version control systems can be modeled with a HIT where committing and reverting correspond to applying equalities of this HIT [AMLH14]. In our project, integrating HITs is also important because they will be used in Objective 5.

Objective 5: Extend homotopy type theory with new logical/computational principles

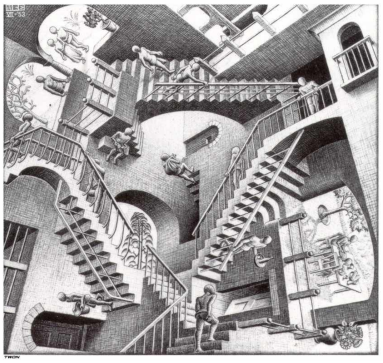
With a practical implementation of HoTT, I plan to reuse well-known model transformations to enhance HoTT with new logical principles. The challenge is to give a meaning, through separate compilation phases, to axioms often used in mathematics, such as the law of excluded middle or the axiom of dependent choice (Fig. 1.1). As adding new logical principles may have a computational cost and may weaken the extraction mechanism, our plan is to extend HoTT modularly, letting the user choose with which extension to work. Thus, this challenge consists in achieving a high level of modularity in the logical principle governing the type theory of the proof assistant. We have already proven [JTS12, JLP+16] that the forcing transformation (which corresponds to the presheaf construction) enables to enhance the logic of Coq with new constructors such as general unrestricted inductive types (not to be confused with higher inductive types). We plan to apply this technique to other constructions that require the univalence axiom to be correct and in particular to the sheaf construction. This will allow for instance to implement in type theory the so-called Gödel translation from classical logic to intuitionistic logic, giving this way a computational meaning to the law of excluded middle.

The structure underlying homotopy type theory is that of ∞-topos recently studied by Jacob Lurie [Lur09]. The main theoretical challenge of this objective will be to make concrete the forcing transformation and the sheaf construction in the setting of ∞-topoi. Indeed, while pre-sheaves and sheaves have already been defined in this setting, the definitions are very abstract and a huge work need to be done to make them sufficiently effective to be implemented using the language of homotopy type theory. For instance, an effective construction of sheaves requires to reformulate sheaves for ∞-topoi using the notion of Lawvere-Tierney topology [MM92]. One of the main issue will be the definition of the associated sheaf functor in that setting.

At this stage, we plan to implement the translation defined in [JTS12, JLP+16] but directly in the language of HoTT. This translation requires some native notion of proof irrelevance for types corresponding to proposition ((-1)-truncated types in the HoTT terminology) that will have to be added beforehand to Coq. We also plan to implement an internalization of the sheaf construction for ∞-topoi. Using those translations, it will be possible to add computationally relevant new logical principles to the core theory. As we expect some significant overhead when using the extended logical setting, some mechanism needs to be found to automatically lift to the extended theory some developments that have been type checked in the core HoTT theory. This way, the overhead will only be paid for modules that actually makes use of the extended principles, which will make our framework more usable in practice.

Chapter 2

Call-by-Name Forcing in Type Theory*



Relativity, Escher (1953)

Contents

*This is joint work with Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot and Matthieu Sozeau [JLP+16].

Forcing has been introduced by Cohen to prove the independence of the Continuum Hypothesis in set theory. The main idea is to build, from a model M, a new model M' for which validity is controlled by a partially-ordered set (poset) of forcing conditions living in M. Technically, a forcing relation $p \Vdash \phi$ between a forcing condition p and a formula ϕ is defined, such that ϕ is true in M' iff $p \Vdash \phi$ is true in M, for some p approximating the new elements of M'. Categorical ideas have been used by Lawvere and Tierney [Tie72] to recast forcing in terms of topos of (pre)sheaves. It is then straightforward to extend the construction to work on categories of forcing conditions, rather than simply posets, giving a proof relevant version of forcing.

Recent years have seen a renewal of interest for forcing, driven by Krivine's classical realizability [Kri09]. In this line of work, forcing is studied as a proof translation, and one seeks to understand its computational content [Miq11b, Bru14], through the Curry-Howard correspondence. This means that $p \Vdash \phi$ is studied as a syntactic translation of formulas, parametrized by a forcing condition p.

Following these ideas, a forcing translation has been defined in [JTS12] for the Calculus of Constructions, the type theory behind the Coq proof assistant. It is based heavily on the presheaf construction of Lawvere and Tierney. The main goal of [JTS12] was to extend the logic behind Coq with new principles, while keeping its fundamental properties: soundness, canonicity and decidability of type checking. This approach can be seen, following [AK16], as type-theoretic metaprogramming.

However, this technique suffers from coherence problems, which complicate greatly the translation. More precisely, the translation of two definitionally equal terms are not in general definitionally equal, but only propositionally equal. Rewriting terms must then be inserted inside the definition of the translation. If this is possible to perform, albeit tedious, when the forcing conditions form a poset, it becomes intractable when we want to define a forcing translation parametrized by a category of forcing conditions.

In this chapter, we propose a novel forcing translation for the Calculus of Constructions (CC_ω), which avoids these coherence problems. Departing from the categorical intuitions of the presheaf construction, it takes its roots in a call-by-push-value [Lev01] decomposition of our system. This will justify to name our translation *call-by-name*, while the previous translation of [JTS12] is *call-by-value*.

"Call-by-name forcing provides the first effectful translation of CC_ω *into itself which preserves definitional equality."*

We then extend our translation to inductive types by exploiting storage operators [Kri94]—an old idea of Krivine to simulate call-by-value in call-by-name in the context of classical realizability—to restrict the power of dependent elimination in presence of effects. The necessity of a restriction should not be surprising and was already present in Herbelin's work [Her12].

This provides *the first version of Calculus of Inductive Constructions (*CIC*) with effects*. The nice property of preservation of definitional equality is emphasized by the implementation of a Coq plugin[1] which works for any term of CIC.

We conclude the chapter by using forcing to produce various results around homotopy type theory. First, we prove that (a simple version of) functional extensionality is preserved in any forcing layer. Then we show that the negation of Voevodsky's univalence axiom is consistent with CIC plus functional extensionality. This statement could already be deduced from the existence of a set-based *proof-irrelevant* model [Wer97], but we provide the first formalization of it, in a proof relevant setting, and by an easy use of the

[1] Available at `https://github.com/CoqHott/coq-forcing`.

forcing plugin. Finally, we show that under an additional assumption of monotonicity of types, we get the preservation of (a simple version of) the univalence axiom.

2.1 Call-by-Push-Value

In this section, we explain how the call-by-push-value language (CBPV) of Levy [Lev01] can be used to present two versions of the forcing translation. To keep our presentation as simple as possible, we will only use a small subset of it, although most of the results can be adapted to a more general setting. The idea of CBPV is to break up the simply-typed λ-calculus, leading to a more atomic presentation distinguishing values and computations, and allowing to add effects easily into the language. We use it as the source language for a generic forcing translation thought of as adding side-effects. Call-by-name and call-by-value strategies can then be decomposed into CBPV, inducing in turn two forcing translations for the λ-calculus.

2.1.1 Syntax of CBPV

CBPV's types and terms are divided into two classes : pure values v and effectful computations t, a dichotomy which is reflected in the typing rules. The syntax and typing rules are given at Figure 2.1

We give some intuition behind those terms. The `thunk` primitive is to be understood as a way of *boxing* a computation into a value. Its dual `force` *runs* the computation. Note that this name has nothing to do with forcing itself and is a coincidence. The `return` primitive creates a pure computation from a value. The `let` binding first evaluates its argument, possibly generating some effects, binds the purified result to the variable and continues with the remaining term. Intuitively, this language is no more than the usual decomposition of a monad into an adjunction.

value types	$A, B ::= \mathcal{U}\, X \mid \alpha$
computation types	$X, Y ::= A \to X \mid \mathcal{F}\, A$
environments	$\Gamma ::= \cdot \mid \Gamma, x : A$
value terms	$v ::= x \mid \texttt{thunk}\ t$
computation terms	$t, u ::= \lambda x : A.\, t \mid t\ v \mid \texttt{let}\ x : A\ \texttt{:=}\ t\ \texttt{in}\ u \mid \texttt{force}\ t \mid \texttt{return}\ v$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_v x : A} \qquad \frac{\Gamma \vdash_c t : X}{\Gamma \vdash_v \texttt{thunk}\ t : \mathcal{U}\, X} \qquad \frac{\Gamma \vdash_v v : \mathcal{U}\, X}{\Gamma \vdash_c \texttt{force}\ v : X}$$

$$\frac{\Gamma, x : A \vdash_c t : X}{\Gamma \vdash_c \lambda x : A.\, t : A \to X} \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_c \texttt{return}\ v : \mathcal{F}\, A}$$

$$\frac{\Gamma \vdash_c t : \mathcal{F}\, A \qquad \Gamma, x : A \vdash_c u : X}{\Gamma \vdash_c \texttt{let}\ x : A\ \texttt{:=}\ t\ \texttt{in}\ u : X} \qquad \frac{\Gamma \vdash_c t : A \to X \qquad \Gamma \vdash_v v : A}{\Gamma \vdash_c t\ v : X}$$

Figure 2.1: Call-by-push-value

For technical reasons, we endow CBPV with reduction rules that are weaker than what is usually assumed, by restricting substitution to strong values, i.e. values which are

not variables, while the standard reduction allows substitution for any value. Indeed, the forcing translation which we present after only interprets this restricted reduction.

Definition 3 (*Restricted CBPV reduction*)

Strong values $\tilde{v}$ are simply defined as $\tilde{v} := \texttt{thunk}\ t$. We define the restricted CBPV reduction as the congruence closure of the following generators.

$$\begin{array}{lcl} (\lambda x : A.\, t)\ \tilde{v} & \to & t\{x := \tilde{v}\} \\ \texttt{let}\ x : A\ \texttt{:=}\ \texttt{return}\ \tilde{v}\ \texttt{in}\ t & \to & t\{x := \tilde{v}\} \\ \texttt{force}\ (\texttt{thunk}\ t) & \to & t \end{array}$$

We write $\equiv$ for the equivalence generated by this reduction.

2.1.2 Simply-Typed Decompositions

We recall here the decompositions of the simply-typed λ-calculus into CBPV. They were actually the original motivation for the introduction of CBPV itself. We will translate the usual λ-calculus where types are described by the inductive grammar

$$A, B := \alpha \mid A \to B$$

using the standard syntax. The results of this section are well-known so we will not dwell on them.

Definition 4

The by-name reduction of the λ-calculus is the congruence closure of the generator

$$(\lambda x : A.\, t)\ u \to_{\mathtt{n}} t\{x := u\}$$

while the restricted by-value reduction is the congruence closure of the generator

$$(\lambda x : A.\, t)\ v \to_{\mathtt{v}} t\{x := v\}$$

where v is a λ-abstraction.

Definition 5 (*By-value decomposition*)

The by-value decomposition is defined as follows.

$$\begin{array}{lcl} [\alpha]^{\mathtt{v}} & := & \alpha \\ [A \to B]^{\mathtt{v}} & := & \mathcal{U}\,([A]^{\mathtt{v}} \to \mathcal{F}\,[B]^{\mathtt{v}}) \\ [x]^{\mathtt{v}} & := & \texttt{return}\ x \\ [t\ u]^{\mathtt{v}} & := & \texttt{let}\ f\ \texttt{:=}\ [t]^{\mathtt{v}}\ \texttt{in} \\ & & \texttt{let}\ x\ \texttt{:=}\ [u]^{\mathtt{v}}\ \texttt{in}\ \texttt{force}\ f\ x \\ [\lambda x : A.\, t]^{\mathtt{v}} & := & \texttt{return}\ (\texttt{thunk}\ (\lambda x : [A]^{\mathtt{v}}.\, [t]^{\mathtt{v}})) \end{array}$$

Proposition 6

If $\Gamma \vdash t : A$ then $[\Gamma]^{\mathtt{v}} \vdash_c [t]^{\mathtt{v}} : \mathcal{F}\,[A]^{\mathtt{v}}$.

Proposition 7

If $t \to_{\mathtt{v}} u$ then $[t]^{\mathtt{v}} \equiv [u]^{\mathtt{v}}$.

The by-name decomposition is defined as follows.

$$\begin{array}{lcl} [\alpha]^{\mathtt{n}} & := & X_\alpha \\ [A \to B]^{\mathtt{n}} & := & \mathcal{U}\,[A]^{\mathtt{n}} \to [B]^{\mathtt{n}} \\ [x]^{\mathtt{n}} & := & \mathtt{force}\ x \\ [t\ u]^{\mathtt{n}} & := & [t]^{\mathtt{n}}\ (\mathtt{thunk}\ [u]^{\mathtt{n}}) \\ [\lambda x : A.\, t]^{\mathtt{n}} & := & \lambda x : \mathcal{U}\,[A]^{\mathtt{n}}.\, [t]^{\mathtt{n}} \end{array}$$

Proposition 8

If $\Gamma \vdash t : A$ *then* $\mathcal{U}\,[\Gamma]^n \vdash_c [t]^n : [A]^n$.

Proposition 9

If $t \to_n u$ *then* $[t]^n \equiv [u]^n$.

2.1.3 Forcing Translation

We now define the forcing translation from CBPV into a small dependent extension of the simply-typed λ-calculus. Dependency is needed because we have to be able to state in the type that some relation holds between two elements. For simplicity, we can use for instance the much richer system defined at Section 2.2. We use implicit arguments and infix notation for clarity when the typing is clear from context.

First of all, we need a notion of preorder in the target calculus.

Definition 10 (*Preorder*)

A preorder is given by

- *a type* $\mathbb{P}$*;*
- *a binary relation* $\leq$*;*
- *a term* $\mathtt{id} : \Pi p : \mathbb{P}.\, p \leq p$*;*
- *a term* $\circ : \Pi(p\,q\,r : \mathbb{P}).\, p \leq q \to q \leq r \to p \leq r$

subject to the following conversion rules.

$$\mathtt{id}_p \circ f \equiv f \quad f \circ \mathtt{id}_q \equiv f \quad f \circ (g \circ h) \equiv (f \circ g) \circ h$$

We assume in the remainder of this section a fixed preorder that we will call *forcing conditions*.

Definition 11 (*Ground types*)

We assume given for every CBPV ground type α*:*

- *a type* α_p *in the target calculus for each* $p : \mathbb{P}$*;*
- *a lifting morphism* $\theta_\alpha : \Pi(p\,q : \mathbb{P}).\, p \leq q \to \alpha_p \to \alpha_q$

subject to the following conversion rules.

$$\theta_\alpha\ \mathtt{id}_p\ x \equiv x \quad \theta_\alpha\ (f \circ g)\ x \equiv \theta_\alpha\ g\ (\theta_\alpha\ f\ x)$$

Definition 12 (*Type translation*)

The forcing translation on types associates to every CBPV type and forcing condition a target type defined inductively as follows.

$$\begin{aligned}
[\![\alpha]\!]_p &:= \alpha_p \\
[\![\mathcal{U}\, X]\!]_p &:= \Pi q : \mathbb{P}.\, p \leq q \to [\![X]\!]_q \\
[\![A \to X]\!]_p &:= [\![A]\!]_p \to [\![X]\!]_p \\
[\![\mathcal{F}\, A]\!]_p &:= [\![A]\!]_p
\end{aligned}$$

Proposition 13 (*Value lifting*)

The lifting morphisms of Definition 11 can be generalized to any value type A as θ_A with the same distribution rules.

Proof: By induction on A. Our only non-variable value type is $\mathcal{U}\, X$ where $\theta_{\mathcal{U}\, X}$ is defined by precomposition. ■

Definition 14 (*Term translation*)

The term translation is indexed by an CBPV environment Γ and a preorder variable p and produces a term in the target calculus. It is defined inductively as

$$\begin{aligned}
[x]^\Gamma_p &:= x \\
[\texttt{thunk}\ t]^\Gamma_p &:= \lambda(q : \mathbb{P})\,(f : p \leq q).\, \theta_\Gamma\ (f, [t]^\Gamma_q) \\
[\texttt{force}\ v]^\Gamma_p &:= [v]^\Gamma_p\ p\ \texttt{id}_p \\
[\lambda x : A.\, t]^\Gamma_p &:= \lambda x : [\![A]\!]_p.\, [t]^{\Gamma, x:A}_p \\
[t\ v]^\Gamma_p &:= [t]^\Gamma_p\ [v]^\Gamma_p \\
[\texttt{let}\ x : A\ \texttt{:=}\ t\ \texttt{in}\ u]^\Gamma_p &:= (\lambda x : [\![A]\!]_p.\, [u]^{\Gamma, x:A}_p)\ [t]^\Gamma_p \\
[\texttt{return}\ v]^\Gamma_p &:= [v]^\Gamma_p
\end{aligned}$$

where the $\theta_\Gamma\ (f, t)$ notation stands for $t\{\vec{x} := \theta_{\vec{A}}\ f\ \vec{x}\}$ for each $(x_i : A_i) \in \Gamma$.

The only non-trivial case of this translation is the `thunk` case, which requires to lift all the free variables of the considered term. We need to do this because the resulting term is *boxed* w.r.t. the current forcing condition by a λ-abstraction, so that there is a mismatch between the free variables of $[t]^\Gamma_q$ which live at level q while we would like them to live at level p. Dually, the `force` translation *resets* a boxed term by applying it to the current condition.

Proposition 15 (*Typing soundness*)

Assume $\Gamma \vdash_c t : X$, then $p : \mathbb{P}, [\![\Gamma]\!]_p \vdash [t]^\Gamma_p : [\![X]\!]_p$ and similarly for values.

Proposition 16 (*Computational soundness*)

For all $\Gamma \vdash_c t,u : A$, if $t \equiv u$ then $[t]^\Gamma_p \equiv [u]^\Gamma_p$ and similarly for values.

The interest of giving this translation directly in CBPV is that we can recover two translations of the λ-calculus by composing it with the by-name and by-value decompositions. This provides hints about the source of the technical impediments encountered in [JTS12].

To start with, we can easily observe that $[\![[A \to B]^{\mathtt{v}}]\!]_p$ is equal to $\Pi q : \mathbb{P}.\, p \leq q \to [\![A]\!]_q \to [\![B]\!]_q$, which is indeed the usual way to translate the arrow type in forcing, as in [JTS12]. The term translation is also essentially the same, except for the adaptations to the dependently-typed case. The two following defects of the call-by-value forcing translation are then obvious through this decomposition.

First, the translation only preserves call-by-value reduction, and not unrestricted β-reduction. Therefore, the interpretation of the conversion rule of CIC by a plain conversion is not possible. One has to resort to more semantical arguments, implying the use of explicit rewriting in the terms.

Second, the very computational conditions imposed over θ_α are highly problematic as soon as we have second-order quantifications. Indeed, we need to ship with each abstracted type $\Pi\alpha : \mathbf{Type}.\, A$ a corresponding θ_α in the translation. But then we loose the definitional equalities required by Definition 11. The only thing we can do is to enforce them by using propositional equalities, which will imply in turn some explicit rewriting.

Meanwhile, the by-name variant is way more convenient to use to interpret CIC conversion. Indeed, it interprets the whole β-conversion, and furthermore it does not even require any θ_α for abstracted variables. This is because all value types appearing in the $[-]^{\mathtt{n}}$ decomposition are of the form $\mathcal{U}\, X$ for some X, so that we statically know we will only need the $\theta_{\mathcal{U}X}$ function which is defined regardless of X. Both properties make a perfect fit for an interpretation of CIC.

2.2 Forcing Translation in the Negative Fragment

In this section, we first consider the forcing translation of CC_ω, a type theory featuring only negative connectives, i.e. Π-types. The syntax and typing rules are given in Appendix.

This translation builds upon the call-by-name forcing described in the previous section. The main differences are that we handle higher-order and dependency, as well as a presentation artifact where we delay the whole-term lifting of the `thunk` translation by using forcing contexts instead. Moreover, we now consider *categories* of forcing conditions, rather than posets.

Definition 17 (*Forcing context*)

Forcing contexts σ are given by the following inductive grammar.

$$\sigma ::= p \mid \sigma \cdot x \mid \sigma \cdot (q, f)$$

In the above definition, p, x, q and f are variables binding in the right of the forcing context, and therefore forcing contexts obey the usual freshness conditions obtained through α-equivalence.

We will often write $\sigma \cdot \varphi$ to represent the forcing context σ extended with some forcing suffix φ made of any kind of extension.

Definition 18 (*Forcing context validity*)

A forcing context σ is valid in a context Γ, written $\Gamma \vdash \sigma$, whenever they pertain to the following inductive relation.

$$\frac{}{\cdot \vdash p} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \cdot (q, f)} \qquad \frac{\Gamma \vdash \sigma}{\Gamma, x : A \vdash \sigma \cdot x}$$

Definition 19 (*Category*)

A category is given by:

- *A term* $\vdash \mathbb{P} : \boldsymbol{Type}_0$ *representing objects;*
- *A term* $\vdash \mathtt{Hom} : \mathbb{P} \to \mathbb{P} \to \boldsymbol{Type}_0$ *representing morphisms;*
- *A term* $\vdash \mathtt{id} : \Pi p : \mathbb{P}.\, \mathtt{Hom}\ p\ p$ *representing identity;*
- *A term* $\vdash \circ : \Pi(p\, q\, r : \mathbb{P}).\, \mathtt{Hom}\ p\ q \to \mathtt{Hom}\ q\ r \to \mathtt{Hom}\ p\ r$ *representing composition.*

For readability purposes, we write $\mathtt{id}_p$ *for* $\mathtt{id}\ p$, $\mathtt{Hom}(p, q)$ *for* $\mathtt{Hom}\ p\ q$ *and we consider the objects for the composition as implicit and write* $f \circ g$ *for* $\circ\ p\ q\ r\ f\ g$ *for some objects* p, q *and* r.

Furthermore, we require that we have the following definitional equalities.

$$\frac{\Gamma \vdash f : \mathtt{Hom}(p, q)}{\Gamma \vdash \mathtt{id}_p \circ f \equiv f} \qquad \frac{\Gamma \vdash f : \mathtt{Hom}(p, q)}{\Gamma \vdash f \circ \mathtt{id}_q \equiv f}$$

$$\frac{\Gamma \vdash f : \mathtt{Hom}(p, q)\ \Gamma \vdash g : \mathtt{Hom}(q, r)\ \Gamma \vdash h : \mathtt{Hom}(r, s)}{\Gamma \vdash f \circ (g \circ h) \equiv (f \circ g) \circ h}$$

Note that asking that they are given definitionally rather than as mere propositional equalities is, as we will see in Section 2.3, actually not restrictive.

Definition 20

The last condition σ_e *from a forcing context* σ *is a variable defined inductively as follows.*

$$p_e := p \quad (\sigma \cdot x)_e := \sigma_e \quad (\sigma \cdot (q, f))_e := q$$

The morphism of a variable x *in a forcing context* σ, *written* $\sigma(x)$, *is a term defined inductively as follows.*

$$p(x) := \mathtt{id}_p \qquad (\sigma \cdot x)(x) := \mathtt{id}_{\sigma_e}$$
$$(\sigma \cdot y)(x) := \sigma(x) \quad (\sigma \cdot (q, f))(x) := \sigma(x) \circ f$$

Notation 21

As it is a recurring pattern in the translation, we will use the following macros.

$$\lambda(q\, f : \sigma).\, M \quad := \quad \lambda(q : \mathbb{P})\, (f : \mathtt{Hom}(\sigma_e, q)).\, M$$
$$\Pi(q\, f : \sigma).\, M \quad := \quad \Pi(q : \mathbb{P})\, (f : \mathtt{Hom}(\sigma_e, q)).\, M$$

Definition 22 (*Forcing translation*)

The forcing translation is inductively defined on terms in Figure 2.2. Note that the three last definitions are simple macros definable in terms of the basic forcing translation that will be used pervasively to ease the reading. In particular, the $[-]^!_\sigma$ *and* $[\![-]\!]^!_\sigma$ *macros correspond respectively to the interpretation of* `thunk` *and* $\mathcal{U}$ *in the call-by-push-value decomposition.*

Assuming that $\Gamma \vdash \sigma$, *which we will do implicitly afterwards, we now define the*

$$
\begin{array}{lcl}
[*]_\sigma & := & \lambda(q\, f : \sigma).\, \Pi(r\, g : \sigma \cdot (q, f)).\, * \\
[\mathbf{Type}_i]_\sigma & := & \lambda(q\, f : \sigma).\, \Pi(r\, g : \sigma \cdot (q, f)).\, \mathbf{Type}_i \\
[x]_\sigma & := & x\ \sigma_e\ \sigma(x) \\
[\lambda x : A.\, M]_\sigma & := & \lambda x : [\![A]\!]^!_\sigma.\, [M]_{\sigma \cdot x} \\
[M\ N]_\sigma & := & [M]_\sigma\ [N]^!_\sigma \\
[\Pi x : A.\, B]_\sigma & := & \lambda(q\, f : \sigma).\, \Pi x : [\![A]\!]^!_{\sigma\cdot(q,f)}.\, [\![B]\!]_{\sigma\cdot(q,f)\cdot x} \\
[\![A]\!]_\sigma & := & [A]_\sigma\ \sigma_e\ \mathtt{id}_{\sigma_e} \\
[M]^!_\sigma & := & \lambda(q\, f : \sigma).\, [M]_{\sigma\cdot(q,f)} \\
[\![A]\!]^!_\sigma & := & \Pi(q\, f : \sigma).\, [\![A]\!]_{\sigma\cdot(q,f)}
\end{array}
$$

Figure 2.2: Forcing translation of the negative fragment.

forcing translation on contexts as follows.

$$
\begin{array}{lcl}
[\![\cdot]\!]_p & := & p : \mathbb{P} \\
[\![\Gamma]\!]_{\sigma\cdot(q,f)} & := & [\![\Gamma]\!]_\sigma, q : \mathbb{P}, f : \mathtt{Hom}(\sigma_e, q) \\
[\![\Gamma, x : A]\!]_{\sigma\cdot x} & := & [\![\Gamma]\!]_\sigma, x : [\![A]\!]^!_\sigma
\end{array}
$$

We now turn to the proof that this translation indeed preserves the typing rules of our theory. As proper typing rules and conversion rules are intermingled, we should actually prove it in a mutually recursive fashion, but this would be fairly unreadable. Therefore, in the following proofs, we rather assume that computational (resp. typing) soundness are already proved for the induction hypotheses, in an open recursion style. This is a mere presentation artifact: the loop is tied at the end by plugging the two soundness theorems together.

Proposition 23 (*Condition Concatenation*)

For any $\Gamma \vdash M : A$, and forcing contexts σ, φ, ψ with φ containing only conditions and morphisms,

$$[\![\Gamma]\!]_{\sigma\cdot\varphi\cdot\psi} \vdash [M]_{\sigma\cdot(q,f)\cdot\psi}\{q := (\sigma \cdot \varphi)_e, f := (\varphi)\} \equiv [M]_{\sigma\cdot\varphi\cdot\psi}$$

where (φ) stands for the composition of all morphisms in φ.

Proof: By induction over M. This property relies heavily on the fact that the categorical equalities are definitional, and the proof actually amounts to transporting those equalities. ■

Proposition 24 (*Substitution Lemma*)

For any $\Gamma \vdash M : A$,

$$[\![\Gamma]\!]_{\sigma\cdot\varphi} \vdash [M\{x := P\}]_{\sigma\cdot\varphi} \equiv [M]_{\sigma\cdot x\cdot\varphi}\{x := [P]^!_\sigma\}$$

Proof: By induction over M and application of the previous lemma. ■

Theorem 25 (*Computational Soundness*)

If $\Gamma \vdash M \equiv N$ then $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma \equiv [N]_\sigma$.

Proof: The congruence rules are obtained trivially, owing to the fact that the translation is defined by induction on the terms. The β-reduction step is obtained by a direct application of the substitution lemma, while the η-expansion rule is interpreted as-is in the translation.

■

Theorem 26 (*Typing Soundness*)

The following holds.

- *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]_\sigma$.
- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma : [\![A]\!]_\sigma$.

Proof: By induction on the typing derivation. The only non-immediate case is the conversion rule which is obtained by applying the computational soundness theorem. ■

Forcing Layer. We now explain how to use the forcing translation to extend safely CIC with new logical principles, so that typechecking remains decidable and the resulting extended theory is equiconsistent with Coq (i.e. if the empty type of Coq, or equivalently the type $\Pi X : \mathbf{Type}.\, X$, is not inhabited then neither is the empty type of the resulting theory) as soon as the type $\mathbb{P}$ of objects is inhabited.

In the *forcing layer*, it is possible to add new symbols to the system. Each symbol $\varphi : \Phi$ has to come with its translation $\vdash \varphi^\bullet : \Pi p : \mathbb{P}.\, [\![\Phi]\!]_p$ in CIC. This is done in the Coq plugin using the command

```
Forcing Definition φ : Φ using ℙ Hom.
```

where $\mathbb{P}$ and `Hom` define the category of forcing conditions in use. Note the similarity with forcing in set theory, where a new model is obtained by adding a generic element G to a ground model, and the forcing relation describes inside the ground model the properties of G in the new model.

The typing relation $\vdash_{\mathcal{F}}$ in the layer is defined by extending CIC with the axiom $\Gamma \vdash_{\mathcal{F}} \varphi : \Phi$. By posing $[\varphi]_\sigma := \varphi^\bullet\ \sigma_e$, it is easy to derive that if $\Gamma \vdash_{\mathcal{F}} M : A$ then $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma : [\![A]\!]_\sigma$ using Theorem 26. The abovementioned equiconsistency result is just a consequence of the fact that if $\vdash_{\mathcal{F}} M : \Pi X : \mathbf{Type}.\, X$ then $\vdash [M]_p : \Pi(X : \Pi(q\ f : p).\, \mathbf{Type}).\, X\, p\, \mathtt{id}$, which shows that a proof of the empty type in the forcing layer directly gives a proof of the empty type in CIC, where the proof term is $\lambda X : \mathbf{Type}.\, [M]_p(\lambda q f.X)$.

2.3 Yoneda to the Rescue

A key property in the preservation of typing is that the forcing category implements category laws in a definitional way. This may seem a very strong requirement. Indeed, any non-trivial operation is going to block on variable arguments, which puts the convertibility at stake. For instance, simply taking objects to be the unit type and morphisms to be booleans equipped with `xor` already breaks at least one of the two identity rules, depending on the order in which `xor` is defined.

Luckily, we can rely on a folklore trick to build for any category an equivalent category with laws that holds definitionally. The soundness of the translation is no more than the good old Yoneda lemma.

Definition 27 (*Yoneda translation*)

Assume a category as given in Definition 19 without assuming any equalities. We define the Yoneda translation of this category as follows.

$$
\begin{array}{lcl}
\mathbb{P}_{\mathcal{Y}} & := & \mathbb{P} \\
\mathtt{Hom}_{\mathcal{Y}}\ p\ q & := & \Pi r : \mathbb{P}.\, \mathtt{Hom}(q, r) \to \mathtt{Hom}(p, r) \\
\mathtt{id}_{\mathcal{Y}}\ p & := & \lambda(r : \mathbb{P})\,(k : \mathtt{Hom}(p, r)).\, k \\
\circ_{\mathcal{Y}}\ p\ q\ r\ f\ g & := & \lambda(s : \mathbb{P})\,(k : \mathtt{Hom}(r, s)).\, g\ s\ (f\ s\ k)
\end{array}
$$

Proposition 28 (*Yoneda lemma*)

The Yoneda translation of a category is a category with laws that holds definitionally.

Proof: Immediate. Typing is straightforward and equalities are simple $\beta\eta$-equivalences. ■

The interesting subtlety of this proof is that we actually do not even need the categorical laws of the base category to recover definitional equalities in the Yoneda translation. What we have done amounts to building the free category generated by objects and morphisms, and definitional equalities follow just because the meta-theory (here, our type theory) is computational. Although the relation between the Yoneda lemma, CPS and free categories has already been observed in the literature, we believe that our current usecase is novel.

It remains now to prove that the Yoneda category is equivalent to its base category. As there is no widely accepted notion of being equivalent in type theory, we are going to allow ourselves to cheat a little bit.

Definition 29 (*Equivalence functors*)

We define two type-theoretic functors $\mathcal{Y}$ (resp. $\mathcal{Y}^{-1}$) between a base category and its Yoneda translation (resp. the converse) as follows. On objects, the translation is the identity

$$\mathcal{Y}_o := \lambda p : \mathbb{P}.\, p \qquad \mathcal{Y}^{-1}_{\,o} := \lambda p : \mathbb{P}_{\mathcal{Y}}.\, p$$

while on morphisms we pose

$$
\begin{array}{lcl}
\mathcal{Y}_h\ p\ q\ f : \mathtt{Hom}_{\mathcal{Y}}\ p\ q & := & \lambda(r : \mathbb{P})\,(k : \mathtt{Hom}(q, r)).\, f \circ k \\
\mathcal{Y}^{-1}_{\,h}\ p\ q\ f : \mathtt{Hom}\ p\ q & := & f\ q\ \mathtt{id}_q
\end{array}
$$

We need to reason about equality, so we suppose until the end of this section that our target type theory features a propositional equality $=$ as defined usually, and furthermore that the functional extensionality principle is provable.

Proposition 30 (*Functoriality*)

Assuming that equalities of Definition 19 hold propositionally, the above objects are indeed functors, i.e. they obey the usual commutation rules w.r.t. identity and composition propositionally.

Proposition 31 (*Category equivalence*)

The above functors form an equivalence in the following sense.

1. *Assuming that equalities of Definition 19 hold propositionally, then* $\mathcal{Y}^{-1}_{\,h}\ p\ q\ (\mathcal{Y}_h\ p\ q\ f) = f$ *propositionally.*
2. *Assuming parametricity over the quantification on the base category, then* $\mathcal{Y}_h\ p\ q\ (\mathcal{Y}^{-1}_{\,h}\ p\ q\ f) = f$ *propositionally.*

Proof: The first equality is straightforward. The second one is essentially an unfolding of the definition of parametricity over the categorical structure. We do not want to dwell too much on the whereabouts of parametricity in this chapter for the lack of space, so that we will not insist on that property and let the reader refer to the actual implementation (see the file `yoneda.v` in the repository `https://github.com/CoqHott/coq-forcing`). ■

Although this is not totally satisfying because of mismatches between type theory and category theory, note that in the special case where the base category is proof-irrelevant (i.e. a preorder) the translation actually builds an equivalent category.

Disregarding these small defects, we will consider that by applying the Yoneda translation to any category, we recover a new category which is essentially the same as the first one except that it has definitional equalities. By plugging it into the forcing translation, we will consequently fulfill all the expected conditions for the soundness theorems to go through.

2.4 Datatypes

We now proceed to extend the calculus with positives, that is datatypes defined by their constructors and move towards a translation of CIC. In CIC, datatypes are defined using a generic schema for declaring inductive types, using a generic eliminator construct for pattern-matching.

We wish to apply the forcing translation to any inductive definition, however there are a number of issues to resolve before doing so, having to do with dependent elimination. For the sake of conciseness, we will focus on Σ-types, whose definition is given in Figure 4.2 of the Appendix. It is noteworthy to remark that we present Σ-types in a positive fashion, that is through pattern-matching, rather than negatively through projections. The latter is usually easier to interpret in an effectful setting, but it is weaker and in general does not extend to other types that have to be interpreted positively such as sums.

Whereas in the plugin our translation of inductive types builds new inductive types, for the sake of simplicity, we will directly translate Σ-types as Σ-types. There is little room left for tinkering. As the translation is by-name, we need to treat the subterms of pairs as application arguments by thunking them using the $[-]^!_\sigma$ macro and similarly for types.

Definition 32 (*Forcing translation of Σ-types*)

$$
\begin{aligned}
[\Sigma x : A.\, P]_\sigma &:= \lambda(q\, f : \sigma).\, \Sigma x : [\![A]\!]^!_{\sigma\cdot(q,f)}.\, [\![P]\!]^!_{\sigma\cdot(q,f)\cdot x} \\
[(M, N)]_\sigma &:= ([M]^!_\sigma, [N]^!_\sigma) \\
[\texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N]_\sigma &:= \texttt{match}\ [M]_\sigma\ \texttt{with}\ (x, y) \Rightarrow [N]_{\sigma\cdot x\cdot y}
\end{aligned}
$$

Proposition 33

The translation enjoys computational soundness.

Against all expectations, typing soundness is not provable for the whole CIC. While the typing rules of formation, introduction and non-dependent elimination are still valid, the dependent elimination rule needs to be restricted. Indeed, the conclusion of the traditional dependent elimination rule for Σ-types is

$$\texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : C\{z := M\}$$

This rule is not valid in presence of effects, because on the left-hand side, M is directly evaluated, whereas on the right-hand side, the evaluation of M is postponed. In particular, it is not valid in the forcing layer, and thus cannot be interpreted by the forcing translation. The translation of this sequent results effectively in

$$\texttt{match}\ [M]_\sigma\ \texttt{with}\ (x,y) \Rightarrow [N]_{\sigma\cdot x\cdot y} : [\![C]\!]_{\sigma\cdot z}\{z := [M]^!_\sigma\}$$

and it is clear that $[M]^!_\sigma$ can have little to do with $[M]_\sigma$. Intuitively, a *boxed* term—i.e. a term expecting a forcing condition before returning a value—of the translated inductive type can use the forcing conditions to build different inductive values at different conditions. It is for instance easy to build boxed booleans, i.e. terms of type $[\![\mathbb{B}]\!]^!_\sigma := \Pi(q\ f : \sigma).\,\mathbb{B}$ that are neither $[\texttt{true}]^!_\sigma$ nor $[\texttt{false}]^!_\sigma$ but whose value depends on the forcing conditions. There is hence no reason for it to be propositionally equal to a constructor application, let alone definitionally.

Therefore, we restrict the source type theory to dependent eliminations where a `match` has type `match`, forcing evaluation in the result type as well. We denote this restricted theory CIC$^-$ and summarize its typing rules at Figure 4.2.

Proposition 34

Typing soundness holds for the CIC$^-$ *rules.*

In this effectful setting, the usual dependent elimination of CIC can be decomposed into a restricted elimination followed by an η-rule for Σ-types which can be written:

$$\texttt{match}\ M\ \texttt{with}\ (x,y) \Rightarrow C\{z := (x,y)\} \equiv_\eta C\{z := M\}.$$

While this η-rule is actually propositionally valid in CIC, it is not preserved by the forcing translation and can be disproved using non-standard boxed terms. In general, assuming definitional η-rules for positive datatypes makes conversion checking hard, in particular for sum types, requiring commutative conversions and very elaborate algorithms even in the simply-typed case [Sch15]. Of course CIC$^-$ plus definitional η-rules for inductive datatypes is equivalent to CIC plus those same rules, but an exact correspondence between CIC$^-$ and CIC is harder to pin down.

Note that the translation also applies directly to the hidden return type annotation found in CIC, which we did not expose here for simplicity. The same technique can be applied to any algebraic datatype.

2.5 Recursive Types

The datatypes described in the previous section are non-recursive. Handling general inductive datatypes raises issues of its own, because we need to be clever enough in the definition to preserve both syntactical typing and reduction rules.

We will define our translation into CIC without giving all the technical details usually imposed by recursive types, amongst others positivity condition and guardedness. The reader can assume a theory close to the one implemented by Coq and Agda for instance. Our practical implementation uses Coq, so that we will use its particular syntax.

Rather than giving the generic translation, which would turn out to be rather uninformative to the reader and too technical, we will focus instead on a running example.[2] This example should be rich enough to uncover the issues stemming from recursive types. We should stick to the `list` type, for it features a parameter. We recall that it is defined as follows.

[2] The Coq plugin translates any (mutually) inductive type.

$$
\begin{array}{l}
\texttt{Inductive list}^\bullet\ (p : \mathbb{P})\ (A : [\![\mathbf{Type}]\!]^!_p)\ : \mathbf{Type} := \\
\mid \texttt{nil}^\bullet : \texttt{list}^\bullet\ p\ A \\
\mid \texttt{cons}^\bullet : [\![A]\!]^!_{p\cdot A} \to [\![\texttt{list}\ A]\!]^!_{p\cdot A} \to \texttt{list}^\bullet\ p\ A
\end{array}
$$

$$
\begin{array}{lcl}
[\texttt{list}\ A]_\sigma & := & \lambda(q\ f : \sigma).\, \texttt{list}^\bullet\ q\ [A]^!_{\sigma\cdot(q,f)} \\
[\texttt{nil}\ A]_\sigma & := & \texttt{nil}^\bullet\ \sigma_e\ [A]^!_\sigma \\
[\texttt{cons}\ A\ M\ N]_\sigma & := & \texttt{cons}^\bullet\ \sigma_e\ [A]^!_\sigma\ [M]^!_\sigma\ [N]^!_\sigma
\end{array}
$$

Figure 2.3: List translation

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

The above definition generates the typing rules below, plus fixpoint and pattern-matching terms with the corresponding rules.

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \texttt{list}\ A : \mathbf{Type}} \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \texttt{nil}\ A : \texttt{list}\ A}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : \texttt{list}\ A}{\Gamma \vdash \texttt{cons}\ A\ M\ N : \texttt{list}\ A}$$

2.5.1 Type and Constructor Translation

On the type itself, the translation is not that difficult. The only really subtle part arises from the forcing translation of types as we have

$$[\![\mathbf{Type}_i]\!]_\sigma := \Pi(q\ f : \sigma).\, \mathbf{Type}_i$$

so that the translation of an inductive type must take a forcing condition and a morphism as arguments.

Now, recursive types appear as arguments of their constructors, and following the by-name discipline, it means that they must be boxed. In particular, it implies that the forcing conditions change at each recursive invocation. There are a lot of possible design choices here when only following typing hints, but only one seems to comply with the reduction rules. It consists in enforcing the fact that the inductive does not depend on the current forcing conditions by simply not taking them as arguments and only rely on one condition.

Formally, we define an intermediate inductive $\texttt{list}^\bullet$, and the forcing translation for the `list` type is derived from it by applying it to the last forcing condition. The whole translation is defined in Figure 2.3. We use macros to show that the translation is straightforward, but they should really be thought of as their unfolding.

Proposition 35 (*Typing soundness*)

The forcing translation of Figure 2.3 preserves the three typing rules of `list`*,* `nil` *and* `cons`*.*

One important remark to do is that even though A is a uniform parameter of the `list` type, it is not anymore in its translation, because it is lifted to a future condition at each recursive call. Indeed, the type $[\![\texttt{list}\ A]\!]^!_{p\cdot A}$ in the recursive call in $\texttt{cons}^\bullet$ is convertible to

$$\Pi(q\ f : p).\, \texttt{list}^\bullet\ q\ (\lambda(r\ g : p \cdot (q, f)).\, A\ r\ (f \circ g))$$

where $\texttt{list}^\bullet$ has a different argument than A. This is not really elegant, but it does not cause more trouble than mere technicalities.

2.5.2 Non-dependent Induction

As in the non-recursive case, it is easy to define a non-dependent recursor on the translation of a recursive inductive type, because pattern-matchings are actually translated as pattern-matchings and similarly for fixpoints. For our running example, we can indeed build a function that folds over a forced list.

Definition 36 (*Recursor*)

A recursor for lists is a term `rec` *of type*

$$T_{\texttt{rec}} := \Pi(A\ P : \textbf{\textit{Type}}).\, P_0 \to P_s \to \texttt{list}\ A \to P$$

with $P_0 := P$ *and* $P_s := A \to \texttt{list}\ A \to P \to P$ *which is subject to the conversions*

$$\begin{aligned} &\texttt{rec}\ A\ P\ H_0\ H_s\ (\texttt{cons}\ A\ M\ N) \equiv H_s\ M\ N\ (\texttt{rec}\ A\ P\ H_0\ H_s\ N) \\ &\texttt{rec}\ A\ P\ H_0\ H_s\ (\texttt{nil}\ A) \qquad\quad \equiv H_0 \end{aligned}$$

assuming the proper typing requirements.

Proposition 37 (*Recursor Translation*)

Assuming a recursor `rec`*, there exists a term* $\texttt{rec}^\bullet$ *of type* $\Pi p : \mathbb{P}.\, [\![T_{\texttt{rec}}]\!]_p$ *such that by posing*

$$[\texttt{rec}\ A\ P\ H_0\ H_s\ M]_\sigma := \texttt{rec}^\bullet\ \sigma_e\ [A]^!_\sigma\ [P]^!_\sigma\ [H_0]^!_\sigma\ [H_s]^!_\sigma\ [M]^!_\sigma$$

the forcing translation interprets the reduction rules of Definition 36 definitionally.

Proof: This recursor is built out of the actual recursor on $\texttt{list}^\bullet$ in a straightforward way. ■

2.5.3 Storage Operators

Just as for the plain datatypes, dependent elimination is troublesome, because non-canonical terms can get in the way. It means that we cannot reasonably aim for the usual induction principles of inductive types, as we can simply disprove them by hand-crafted terms. The situation is actually even direr, because trying to take a simple `match`-expansion trick is not enough to make the inductive case go through. We need something stronger.

Luckily, we came up with a restriction inspired from another context where forcing interacts with effects: classical realizability. In order to recover the induction principle on natural numbers in presence of `callcc`, Krivine introduced the notion of *storage operators* [Kri94]. Essentially, a storage operator, *e.g.*, for integers, is a term $\vartheta_{\mathbb{N}}$ of type $\mathbb{N} \to (\mathbb{N} \to R) \to R$ which purifies an integer argument by recursively evaluating and reconstructing it. The suspicious $(\mathbb{N} \to R) \to R$ return type is actually a trick to encode call-by-value in a call-by-name setting thanks to a CPS, so that we are sure that the integer passed to the continuation is actually a value.

Storage operators are somehow arcane outside of the realm of classical realizability, but they are actually both really simple to define from a recursor, computationally straightforward and invaluable to our translation. Once again, we only define here a storage operator for the `list` type but this can be generalized.

$$\begin{array}{rcl}\vartheta & : & \Pi(A\,R : \mathbf{Type}).\,\texttt{list}\ A \to (\texttt{list}\ A \to R) \to R \\ & := & \lambda(A\,R : \mathbf{Type}).\,\texttt{rec}\ A\ ((\texttt{list}\ A \to R) \to R) \\ & & \quad (\lambda k : \texttt{list}\ A \to R.\,k\ (\texttt{nil}\ A)) \\ & & \quad (\lambda(x : A)\,(\_ : \texttt{list}\ A)\,(r : (\texttt{list}\ A \to R) \to R)\,(k : \texttt{list}\ A \to R). \\ & & \quad\quad r\ (\lambda l : \texttt{list}\ A.\,k\ (\texttt{cons}\ A\ x\ l)))\end{array}$$

Figure 2.4: Storage operator for lists

Definition 38 (*Storage operator*)

Assuming a recursor `rec`*, we define the storage operator for lists* ϑ *in Figure 2.4. We will omit the* A *and* R *arguments when applying* ϑ *for brevity.*

Storage operators are only defined by means of the non-dependent recursor, so they have a direct forcing translation by applying Proposition 37. Moreover, in a pure setting, they are pretty much useless, as the following proposition holds.

Proposition 39 (*Propositional η-rule*)

CIC *proves that*

$$\Pi(A\,R : \mathbf{Type})\,(l : \texttt{list}\ A)\,(k : \texttt{list}\ A \to R).\,\vartheta\ l\ k = k\ l.$$

This is proved by a direct *dependent* induction over the list. This is precisely where the forcing translation fails, and the above theorem does not survive the forcing translation.

2.5.4 Dependent Induction in an Effectful World

By using storage operators, we can nevertheless provide the effectful equivalent of an induction principle on recursive types.

Theorem 40

There exists a term $\texttt{ind}^\bullet$ *of type* $\Pi p : \mathbb{P}.\,[\![T_{\texttt{ind}}]\!]_p$ *where*

$$\begin{array}{rcl}T_{\texttt{ind}} & := & \Pi(A : \mathbf{Type})\,(P : \texttt{list}\ A \to \mathbf{Type}). \\ & & \quad P_0 \to P_s \to \Pi l : \texttt{list}\ A.\,\vartheta\ l\ P \\ P_0 & := & P\ (\texttt{nil}\ A) \\ P_s & := & \Pi(x : A)\,(l : \texttt{list}\ A).\,\vartheta\ l\ P \to \vartheta\ (\texttt{cons}\ A\ x\ l)\ P\end{array}$$

which is subject to the conversion rules of Definition 36 (by replacing `rec` *by* `ind`*).*

Proof: Once again, it is a straightforward application of the dependent induction principle for $\texttt{list}^\bullet$. ■

In the usual CIC, the above theorem seems to be a very contrived way to state the dependent induction principle. By rewriting the propositional η-rule, even its type is *equal* to the type of the usual induction principle. Yet, in the effectful theory resulting from the forcing translation, the two theorems are sharply distinct, as the usual induction principle is disprovable in general.

2.5.5 Revisiting the Non-Recursive Case

Actually, even the restriction on dependent elimination from Section 2.4 can be presented in terms of storage operators. As soon as a non-recursive type is defined by constructors, one can easily define storage operators over it by pattern-matching alone.

$$\begin{array}{rcl} \vartheta_\Sigma & : & \Pi(A : \mathbf{Type})\,(B : A \to \mathbf{Type})\,(R : \mathbf{Type}).\,(\Sigma x : A.\,B) \to ((\Sigma x : A.\,B) \to R) \to R \\ & := & \lambda(A : \mathbf{Type})\,(B : A \to \mathbf{Type})\,(R : \mathbf{Type})\,(p : \Sigma x : A.\,B)\,(k : (\Sigma x : A.\,B) \to R). \\ & & \quad \texttt{match}\ p\ \texttt{with}\ (x, y) \Rightarrow k\ (x, y) \end{array}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\,B \qquad \Gamma, z : \Sigma x : A.\,B \vdash C : \mathbf{Type} \qquad \Gamma, x : A, y : B \vdash N : C\{z := (x, y)\}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : \vartheta_\Sigma\ M\ (\lambda z : \Sigma x : A.\,B.\,C)}$$

Figure 2.5: Storage operator for Σ-types

Definition 41 (*Simple storage operator*)

We define a storage operator ϑ_Σ for Σ-types in Figure 2.5.

It is now obvious that the `match` restriction when typing dependent pattern-matching corresponds exactly to the insertion of a storage operator in front of the type of the expression, i.e. the typing rule of Figure 2.5 is equivalent to the one of Section 2.4 up to conversion.

Therefore, we advocate for the use of storage operators as a generic way to control effects in a dependent setting. Purity is recovered by adding the η-law on storage operators as a theorem in the theory, or even definitionally. To the best of our knowledge, this use of storage operators is novel in a dependent type theory equipped with effects, notwithstanding the actual existence of such an object.

2.6 Forcing at Work: Consistency Results

In this section, we present preservation of (a simple version of) functional extensionality and the fact that the negation of the univalence axiom is compatible with CIC. Then, we show that (a simple version of) the univalence axiom is preserved for types which respect a monotonicity condition.

2.6.1 Equality in CIC

Before stating consistency result, we need to look at the notion of equality in CIC and in the forcing layer. As usual, equality in CIC is given by the inductive `eq` with one constructor `refl` as follows:

```
Inductive eq(A : Type)(x : A) : A → Type :=
| refl : eq A x x
```

We simply write $x = y$ for `eq` $A\ x\ y$ when A is clear from context. Following the automatic translation of inductive types explained in Section 2.5, `eq` is translated as

$$\begin{array}{l} \texttt{Inductive}\ \texttt{eq}^\bullet\,(p : \mathbb{P})\,(A : [\![\mathbf{Type}]\!]^!_p)\,(x : [\![A]\!]^!_p)\ : [\![A]\!]^!_p \to \mathbf{Type} := \\ \mid \texttt{refl}^\bullet : \texttt{eq}^\bullet\ p\ A\ x\ x \end{array}$$

Actually, we can show that the canonical function from $x = y$ to $\texttt{eq}^\bullet\ p\ A\ x\ y$ (obtained by eliminating over $x = y$) is an equivalence[3] for all forcing condition p. This means that the property satisfied by `eq` in the core calculus can be used to infer properties on `eq` in the forcing layer.

Using a storage operator, we can define a dependent elimination that corresponds to the J eliminator of Martin-Löf's type theory. Nevertheless, we simply need here the following Leibniz principle, which avoids the use of storage operators because the returned predicate does not depend on the equality:

$$\Pi A\,(x\,y : A)\,(P : A \to \mathbf{Type})\,(e : x = y).\,P\ x \to P\ y.$$

2.6.2 Preservation of Functional Extensionality

The precise statement of functional extensionality in homotopy type theory is that the term `f_equal` of type:

$$\Pi A\,(B : A \to \mathbf{Type})\,(\varphi\,\psi : \Pi x.\,B\ x).\,\varphi = \psi \to \Pi x.\,\varphi\ x = \psi\ x$$

is an equivalence. This term is obtained from Leibniz's principle and expresses that when two functions are equal, they are equal pointwise.

Assuming functional extensionality in the core calculus, we can define a weaker variant of functional extensionality.

Proposition 42 (*Preservation of functional extensionality*)

Assuming functional extensionality in the core calculus, one can define a term of type

$$\Pi A\,(B : A \to \mathbf{Type})\,(\varphi\,\psi : \Pi x.\,B\ x).\,(\Pi x.\,\varphi\ x = \psi\ x) \to \varphi = \psi$$

in the forcing layer.

Proof: Once translated in the core calculus, using the equivalence between `eq` and $\texttt{eq}^\bullet$, it remains to give a term of type $\varphi = \psi$ for all forcing condition p and φ and ψ in $[\![\Pi x : A.\,B\ x]\!]_p$, assuming a term X of type $[\![\Pi x.\,\varphi\ x = \psi\ x]\!]^!_p$. Now, φ and ψ are functions that expect a forcing condition q, a morphism $f : \texttt{Hom}\ p\ q$ and an argument $[\![A]\!]^!_{p\cdot(q,f)}$. Using functional extensionality in the core calculus, this amounts to show $\varphi\ q\ f\ x = \psi\ q\ f\ x$. This can be deduced by using `f_equal` on $X\ p\ q\ x$ and applying it to q and `id`. ■

The preservation of the complete axiom of functional extensionality would require some additional naturality conditions (similar to parametricity) in the translation (see Section 2.6.5 for a discussion on this point).

In the same way, we can prove the preservation of the *Uniqueness of Identity Proof* axiom which says that any proof of $x = x$ is by reflexivity.

2.6.3 Negation of the Univalence Axiom

In homotopy type theory, Voevodsky's univalence axiom is expressed by saying that the canonical map `path_to_equiv` of type

$$A = B \to \Sigma\varphi : A \to B.\,\texttt{IsEquiv}\ A\ B\ \varphi$$

[3] In homotopy type theory, being an equivalence is defined as the predicate

$$\begin{aligned}\texttt{IsEquiv} := \lambda(A\,B : \mathbf{Type})\,(f : A \to B).\\ \Sigma g : B \to A.\,(\Pi x.\,g\ (f\ x) = x) \times (\Pi y.\,f\ (g\ y) = y).\end{aligned}$$

is an equivalence. This term is defined using Leibniz's principle on the identity equivalence. It is a very important axiom, which sheds light on the connection between CIC and homotopy theory—more specifically higher topos theory. This axiom expresses that the only way to observe a type is through its interaction with the environment. Actually, this axiom can be wrong in presence of effects because types may perform effects that cannot be observed because a type A is always observed uniformly at every possible future condition and not at a given one.

Proposition 43 (*Negation of the univalence axiom*)

There exists a forcing layer in which the type

$$(\Pi(A\,B : \mathbf{Type}).\, \texttt{IsEquiv}\ \_\ \_\ (\texttt{path\_to\_equiv}\ A\ B)) \to \bot$$

can be inhabited (where $\bot$ is the inductive type with no constructor).

Proof: We define the forcing condition to be $\mathbb{P} := \texttt{bool}$ and for all $p, q : \texttt{bool}$, $\texttt{Hom}(p,q) := \texttt{unit}$ where `bool` (resp. `unit`) is the inductive type with two (resp. one) elements. In this layer, it is possible to define two new types (at level p)

$$\begin{array}{rcll} A_0 & := & \lambda(q\,f : p).\, \texttt{if}\ q\ \texttt{then unit else}\ \bot & : \quad [\![\mathbf{Type}]\!]_p \\ A_1 & := & \lambda(q\,f : p).\, \texttt{if}\ q\ \texttt{then}\ \bot\ \texttt{else unit} & : \quad [\![\mathbf{Type}]\!]_p \end{array}$$

Those two types are obviously different in the forcing layer. However, it is possible to define a function from A_0 to A_1 by using the fact that functions expect their arguments to be given for every possible future forcing condition. Thus, to define the function at condition, say p, one just have to use the argument at condition $\neg p$, the negation of p. Symmetrically, it is possible to define a function from A_1 to A_0, and to show that they form an equivalence. ■

Note that the univalence axiom has been shown to be consistent with Martin-Löf's type theory using a simplicial model [KLV12], which suggests the independence of the univalence axiom with CIC.

2.6.4 Preserving Univalence Axiom for Monotonous Types

In the previous section, we have been able to negate the univalence axiom by using types that produce completely non-monotonous effects. But if we restrict the univalence statement to types that respect a monotonicity condition, it becomes possible to prove the preservation of (a simple version of) univalence. Indeed, it is possible to define a modality $\circledcirc$ on **Type** by

$$\circledcirc_p : [\![\mathbf{Type} \to \mathbf{Type}]\!]_p := \lambda X\, q\, f.\, \Pi r\, (g : \texttt{Hom}\ q\ r).\, X\ r\ (g \circ f)\ r\ \texttt{id}$$

We get a modality in the sense of [Uni13][4]. A type A is $\circledcirc$-modal when it is equivalent to $\circledcirc A$. Those types are the types which satisfy a monotonicity condition. Restricting the univalence axiom to $\circledcirc$-modal types, we can recover (a simple form of) preservation of univalence.

Proposition 44 (*Preservation of the univalence axiom for $\circledcirc$-modal types*)

Assuming univalence in the core calculus, one can define a term of type

$$(\Sigma\varphi : A \to B.\, \texttt{IsEquiv}\ A\ B\ \varphi) \to \circledcirc A = \circledcirc B$$

in the forcing layer.

[4] Up to a missing equality that can be recovered using naturality conditions of Section 2.6.5

Proof: The proof is similar to the proof of preservation of functional extensionality. It also uses the fact that assuming univalence in the core calculus also implies functional extensionality in the core calculus. The crux of the proof lies in the fact that A and B have only to be equal globally, and not pointwisely at each forcing condition.

For instance, the types A_0 and A_1 of Proposition 43 satisfy $(\circledcirc A_0) = (\circledcirc A_1)$. ■

2.6.5 Towards Forcing with Naturality Conditions

Our forcing translation is much coarser than it could be, for it allows really non-standard terms that can abuse the forcing conditions a lot. Most notably, all boxed terms coming from the translation respect strong constraints that the current translation does not account for, and which are the call-by-name equivalent to the *naturality* requirement from the presheaf construction. For instance, all closed boxed types $A^\bullet : [\![\mathbf{Type}]\!]^!_\sigma \equiv \Pi(q\, f : \sigma)\,(r\, g : \sigma \cdot (q, f)).\, \mathbf{Type}$ verify the equality

$$A^\bullet\ q\ f\ r\ g \equiv A^\bullet\ r\ (f \circ g)\ r\ \mathtt{id}_r$$

for all q, f, r and g. The same goes for inductive types, as the need to restrict dependent elimination in CIC$^-$ stems from the existence of boxed terms that allow themselves to observe the current conditions too much. By enforcing the fact that they must coincide at each later condition, we could recover a propositional η-rule and thus full dependent elimination.

Actually, it seems not that difficult to enforce such naturality properties by means of an additional bit of parametricity in the translation itself, in the style of Lasson [BL11]. Just as the call-by-value translation requires natural propositional equalities on the value types, we can do the same for values appearing in the CBPV decomposition of call-by-name, i.e. in the $[-]^!_\sigma$ and $[\![-]\!]^!_\sigma$ translations. This also means that the translation of each type A must embed a parametricity property $\Vdash_{A,\sigma} : [\![A]\!]^!_\sigma \to \mathbf{Type}$ specifying what it is to be natural at this type (i.e. parametric).

We believe that contrarily to the call-by-value forcing, this should not prevent the translation to preserve definitional equality. Indeed, as in the parametricity translation of PTS, we never rely on the additional equalities to compute, and merely pass them along the translation. Even more, the unary parametricity translation should probably be equivalent to the forcing translation with trivial conditions.

Such a translation would be in some sense purer. It would preserve the monotonous univalence axiom from the previous section, but also allow to prove propositionally the η-law for storage operators. Therefore, it would be the best of by-value and by-name forcing translations.

2.7 Future Work

Our work allows to use any category to increase the logical power of CIC just as considering presheaves allows to increase the logical power of a topos. This is a first step towards the use of the category of cubes as the type of forcing conditions to give a computational content to the cubical type theory [CCHM16] of Coquand et al and in particular to the univalence axiom.

It also shed some new light on the difficult problem of combining dependent types with effects. Indeed, our translation is really close to a reader monad, the forcing conditions corresponding to some states that can be read, and locally modified in a monotonic way. It would be interesting to see if some of the techniques introduced here, notably the use of storage operators, could be applied to handle more general effects.

Chapter 3

Chemical Foundations of Distributed Aspects

Faisceaux du phare du pilier, Noirmoutier

Contents

Distributed applications are complex to develop because of a plethora of issues related to synchronization, distribution, and mobility of code and data across the network. It has been advocated that traditional programming languages do not allow to separate distribution concerns from standard functional concerns in a satisfactory way. For instance, data replication, transactions, security, and fault tolerance often crosscut the business code of a distributed application. Aspect-Oriented Programming (AOP) promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns [KIL+96, EFB01]. Indeed, the pointcut/advice mechanism of AOP provides the facility to intercept the flow of control when a program reaches certain execution points (called *join points*) and perform new computation (called *advice*). The join points of interest are denoted by a predicate called a *pointcut*.

AOP is frequently used in distributed component infrastructures such as Enterprise Java Beans, application frameworks (such as Spring[1]) and application servers (such as JBoss[2]). Recently, there is a growing interest in the use of AOP for Cloud computing [MBHJJ11, Che12], including practical infrastructures such as CloudStack[3]. In all these cases however, AOP systems do not support the remote definition or application of aspects. Rather, non-distributed aspects are used to manipulate distributed infrastructures [SLB02].

To address these limitations, distributed AOP has been the focus of several practical developments: JAC [PSD+04], DJcutter [NCT04], QuO's ASL [DLS+04], ReflexD [TT06], AWED [BNSV+06, BNSVV06], Lasagne [TJ06], as well as a higher-order procedural language with distribution and aspects [TFD+10]. These languages introduce new concepts for distributed AOP such as remote pointcut (advice triggered by remote join points), distributed advice (advice executed on a remote host), migration of aspects, asynchronous and synchronous aspects, distributed control flow, etc. Most of these systems are based on Java and RMI in order to promote the role of AOP on commonly-used large-scale distributed applications. But the temptation of using a rich language to develop interesting applications has the drawback that it makes it almost impossible to define the formal semantics of distributed aspects. While the formal foundations of aspects have been laid out in the sequential setting [WKD04, DFES10], to date, no theory of distributed aspects has been developed.[4]

This chapter develops the formal foundations of distributed AOP using a chemical calculus, essentially a variant of the distributed join calculus [FG96b]. The join calculus is a functional name-passing calculus based on the chemical abstract machine and implemented in several mainstream languages like OCaml [FLFMS03], C# [BCF04] and Scala [HVC08]. Chemical execution engines are also being developed for Cloud computing [PPT11, OT13]. Due to its chemical nature, the join calculus is well-suited to describe parallel computation. The explicit treatment of localities and migration in the distributed join calculus make it possible to express distribution-related concerns directly.

In the join calculus, communication channels are created together with a set of reaction rules that specify, once and for all, how messages sent on these names are synchronized and processed. The crosscutting phenomena manifests in programs written in this style, just as they do in other languages. The reason is that reactions in the join calculus

[1] `http://www.springsource.org`

[2] `http://www.jboss.org`

[3] `http://cloudstack.apache.org/`

[4] This chapter builds upon the conference publication [Tab10]. Much of the text has been completely rewritten with Éric Tanter. The aspect join calculus has been simplified and clarified, in particular by removing the type system and the management of classes, because they are orthogonal to the extensions considered in this work. The implementation is available at `http://tabareau.fr/aspect_jocaml/`.

are scoped: it is not possible to define a reaction that consumes messages on external channels. Therefore, extending a cache process with replication implies modifying the cache definition itself. Similarly, establishing alternative migration policies based on the availability of locations requires intrusively modifying components.

The Aspect Join Calculus developed in this chapter addresses crosscutting issues in the join calculus by introducing the possibility to define aspects that can react to chemical reactions. In doing so, it provides a formal foundation that can be used to understand and describe the semantics of existing and future distributed aspect languages. We also use it to describe interesting features that have not (yet) been implemented in practical distributed AOP systems.

3.1 The distributed objective join calculus

We start by presenting a distributed and object-oriented version of the join calculus.[5] This calculus, which we call the *distributed objective join calculus*, is an original, slightly adapted combination of an object-oriented version of the join calculus [FLMR03] and an explicit notion of location to account explicitly for distribution [FG02].

3.1.1 Message passing and internal states

Before going into the details of the distributed objective join calculus, we begin with the example of the object *buffer* presented in [FLMR03]. The basic operation of the join calculus is asynchronous message passing and, accordingly, the definition of an object describes how messages received on some labels can trigger processes. For instance, the term

$$\mathsf{obj}\ r = \mathit{reply}(n) \triangleright \mathit{out}.\mathit{print}(n)$$

defines an object that reacts to messages on its own label *reply* by sending a message with label *print* and content n to an object named *out* that prints on the terminal. In the definition of an object, the '$\triangleright$' symbol defines a reaction rule that consumes the messages on its left hand side and produces the messages on its right hand side.

Note that labels may also be used to represent the internal state of an object. Consider for instance the definition of a one-place buffer object:

$$\begin{array}{rll}\mathsf{obj}\ b = & & \mathit{put}(n)\ \&\ \mathit{empty}() \triangleright b.\mathit{some}(n) \\ & \mathsf{or} & \mathit{get}(r)\ \&\ \mathit{some}(n) \triangleright r.\mathit{reply}(n)\ \&\ b.\mathit{empty}() \\ \mathsf{in}\ b.\mathit{empty}() & & \end{array}$$

A buffer can either be empty or contain one element. The buffer state is encoded as a message pending on *empty* or *some*, respectively. A buffer object is created empty, by sending a first message $b.\mathit{empty}$ in the in clause. Note that to keep the buffer object consistent, there should be a single message pending on either *empty* or *some*. This remains true as long as external processes cannot send messages on these internal labels directly. This can be enforced by a privacy discipline, as described in [FLMR03].

[5] There is a good reason why we choose a variant of the join calculus with objects; we discuss it later in Section 3.2.4, once the basics of aspects in the calculus are established.

$P \stackrel{\text{def}}{=}$		**Processes**
	0	null process
	$x.M$	message sending
	$\mathsf{obj}\ x = D\ \mathsf{in}\ P$	object definition
	$\mathsf{go}(H); P$	migration request
	$H[P]$	situated process
	$P \,\&\, P$	parallel composition
$D \stackrel{\text{def}}{=}$		**Definitions**
	$M \triangleright P$	reaction rule
	$D\ \mathsf{or}\ D$	disjunction
$M \stackrel{\text{def}}{=}$		**Patterns**
	$l(\bar{v})$	message
	$M \,\&\, M$	synchronization
$\mathcal{D} \stackrel{\text{def}}{=}$		**Named Definitions**
	$x.D$	object definition
	$H[\mathcal{D} : P]$	sub-location definition
	$\mathcal{D}\ \mathsf{or}\ \mathcal{D}$	disjunction
	$\top$	void definition

Figure 3.1: Syntax of the distributed objective join calculus (a combination of simplified versions of the distributed join calculus [FG02] and the objective join calculus [FLMR03])

3.1.2 Syntax

We use three disjoint countable sets of identifiers for object names $x, y, z \in \mathcal{O}$, labels $l \in \mathcal{L}$ and host names $H \in \mathcal{H}$. Tuples are written $(v_i)^{i \in I}$ or simply $\bar{v}$. We use v to refer indifferently to object or host names, *i.e.*, $v \in \mathcal{O} \bigcup \mathcal{H}$. The grammar of the distributed objective join calculus is given in Figure 3.1; it has syntactic categories for processes P, definitions D, patterns M, and named definitions $\mathcal{D}$. The main construct is object definition $\mathsf{obj}\ x = D\ \mathsf{in}\ P$ that binds the name x to the definitions of D. The scope of x is every guarded process in D (here x means "self") and the process P. Objects are taken modulo renaming of bound names (or α-conversion).

Definitions D are a disjunction of reaction rules. A reaction rule $M \triangleright P$ associates a pattern M with a guarded process P. Every message pattern $l(\bar{v})$ in M binds the object names and/or hosts $\bar{v}$ with scope P. In the join calculus, it is required that every pattern M guarding a reaction rule be linear, that is, labels and names appear at most once in M. Named definitions $\mathcal{D}$ are a disjunction of object definitions $x.D$ and sub-location definitions $H[\mathcal{D} : P]$, hosting the named definitions $\mathcal{D}$ and process P at host H. Note that each object is associated to exactly one named definition. $H[P]$ is the process that starts a fresh new location with process P. Note that $H[P]$ acts as a binder for creating a new host. A migration request is described by $\mathsf{go}(H'); P$. It is subjective in that it provokes the migration of the current host H to any location of the form $\psi H'$ (which must be unique by construction) with continuation process P. The definitions of free names (noted $\mathsf{fn}(.)$) for processes, definitions, patterns and named definitions are given in Figure 3.2.

$\mathsf{fn}(0)$	$=$	$\emptyset$
$\mathsf{fn}(x.M)$	$=$	$\{x\} \cup \mathsf{fn}(M)$
$\mathsf{fn}(\mathsf{obj}\ x = D\ \mathsf{in}\ P)$	$=$	$(\mathsf{fn}(D) \cup \mathsf{fn}(P)) \setminus \{x\}$
$\mathsf{fn}(\mathsf{go}(H); P)$	$=$	$\{H\} \cup \mathsf{fn}(P)$
$\mathsf{fn}(H[P])$	$=$	$\mathsf{fn}(P) \setminus H$
$\mathsf{fn}(P \,\&\, Q)$	$=$	$\mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$\mathsf{fn}(M \triangleright P)$	$=$	$\mathsf{fn}(P) \setminus \mathsf{fn}(M)$
$\mathsf{fn}(D\ \mathsf{or}\ D')$	$=$	$\mathsf{fn}(D) \cup \mathsf{fn}(D')$
$\mathsf{fn}(l(\bar{v}))$	$=$	$\{v_i / i \in I\}$
$\mathsf{fn}(M \,\&\, M')$	$=$	$\mathsf{fn}(M) \cup \mathsf{fn}(M')$
$\mathsf{fn}(x.D)$	$=$	$\{x\} \cup \mathsf{fn}(D)$
$\mathsf{fn}(H[\mathcal{D} : P])$	$=$	$(\mathsf{fn}(\mathcal{D}) \cup \mathsf{fn}(P)) \setminus H$
$\mathsf{fn}(\mathcal{D}\ \mathsf{or}\ \mathcal{D}')$	$=$	$\mathsf{fn}(\mathcal{D}) \cup \mathsf{fn}(\mathcal{D}')$
$\mathsf{fn}(\top)$	$=$	$\emptyset$

Figure 3.2: Definition of free names $\mathsf{fn}(\cdot)$

3.1.3 Semantics

The operational semantics of the distributed objective join calculus is given as a *reflexive chemical abstract machine* [FG96b]. A machine $\mathcal{D} \Vdash^{\varphi} \mathcal{P}$ consists in a set of named definitions $\mathcal{D}$ and of a multiset of processes $\mathcal{P}$ running in parallel at location $\varphi = H_1 \cdots H_n$. Each rewrite rule applies to a configuration $\mathcal{C}$, called a *chemical solution*, which is a set of machines running in parallel:

$$\mathcal{C} = \mathcal{D}_1 \Vdash^{\varphi_1} \mathcal{P}_1 \quad \| \quad \cdots \quad \| \quad \mathcal{D}_n \Vdash^{\varphi_n} \mathcal{P}_n$$

Intuitively, a root location H can be thought of as an IP address on a network and a machine at host/root location H can be thought of as a physical machine at this address. Differently, a machine at sub-location HH' can be thought of as a system process H' executing on a physical machine (whose location is H). This includes for example the treatment of several threads, or of multiple virtual machines executing on the same physical machine.

A chemical reduction is the composite of two kinds of rules: (i) structural rules $\equiv$ that deal with (reversible) syntactical rearrangements, (ii) reduction rules $\longrightarrow$ that deal with (irreversible) basic computational steps. The rules for the distributed objective join calculus are given in Figure 3.3. In chemical semantics, each rule is local in the sense that it mentions only messages involved in the reaction; but it can be applied to a wider chemical solution that contains those messages. By convention, the rest of the solution, which remains unchanged, is implicit.

Rules Or and Empty make composition of named definitions associative and commutative, with unit $\top$. Rules Par and Nil do the same for parallel composition of processes. Rule Join gathers messages that are meant to be matched by a reaction rule. Rule Obj-Def describes the introduction of an object (up-to α-renaming, we can consider that any definition of an object x appears only once in a configuration). The reduction rule Red specifies how a message $x.M'$ interacts with a reaction rule $x.[M \triangleright P]$. The notation $x.[M \triangleright P]$ means that the unique named definition $x.D$ in the solution contains reaction rule $M \triangleright P$. The message $x.M'$ reacts when there exists a substitution σ with domain $\mathsf{fn}(M)$ such that $M\sigma = M'$. In that case, $x.M\sigma$ is consumed and replaced by a copy of the substituted guarded process $P\sigma$.

Structural rules

OR
$(\mathcal{D} \text{ or } \mathcal{D}') \Vdash^{\varphi} \equiv \mathcal{D}, \mathcal{D}' \Vdash^{\varphi}$

EMPTY
$\top \Vdash^{\varphi} \equiv \Vdash^{\varphi}$

PAR
$\Vdash^{\varphi} P \,\&\, Q \equiv \Vdash^{\varphi} P, Q$

NIL
$\Vdash^{\varphi} 0 \equiv \Vdash^{\varphi}$

JOIN
$\Vdash^{\varphi} x.(M \,\&\, M') \equiv \Vdash^{\varphi} x.M, x.M'$

SUB-LOC
$H[\mathcal{D} : P] \Vdash^{\varphi} \equiv \{\mathcal{D}\} \Vdash^{\varphi H} \{P\}$ (H frozen)

OBJ-DEF
$\Vdash^{\varphi} \mathsf{obj}\ x = D\ \mathsf{in}\ P \equiv x.D \Vdash^{\varphi} P$ (x fresh)

LOC-DEF
$\Vdash^{\varphi} H[P] \equiv H[\top : P] \Vdash^{\varphi}$ (H fresh)

Reduction rules

RED
$x.[M \triangleright P] \Vdash^{\varphi} x.M\sigma \longrightarrow x.[M \triangleright P] \Vdash^{\varphi} P\sigma$

MESSAGE-COMM
$\Vdash^{\varphi} x.M \parallel x.D \Vdash^{\psi} \longrightarrow \Vdash^{\varphi} \parallel x.D \Vdash^{\psi} x.M$

MOVE
$H[\mathcal{D} : (P \,\&\, \mathsf{go}(H'); Q)] \Vdash^{\varphi} \parallel \Vdash^{\psi H'} \longrightarrow \Vdash^{\varphi} \parallel H[\mathcal{D} : (P \,\&\, Q)] \Vdash^{\psi H'}$

Figure 3.3: Chemical semantics of the distributed objective join calculus (adapted from [FG02, FLMR03])

Distribution. Rule MESSAGE-COMM states that a message emitted in a given location φ on a channel name x that is remotely defined can be forwarded to the machine at location ψ that holds the definition of x. Later on, this message can be used within ψ to assemble a pattern of messages and to consume it locally, using a local RED step. Note that in contrast to some models of distributed systems [RH98], the routing of messages is not explicitly described by the calculus.

Rule LOC-DEF describes the introduction of a sub-location (up-to α-conversion, we can consider that any host appears only once in a configuration). Rule SUB-LOC introduces a new machine at sub-location φH of φ with $\mathcal{D}$ as initial definitions and P as initial process. When read from right-to-left, the rule can be seen as a serialization process, and conversely as a deserialization process. The side condition "H frozen" means that there is no other machine of the form $\Vdash^{\varphi H\psi}$ in the configuration (*i.e.*, all sub-locations of H have already been "serialized"). The notation $\{\mathcal{D}\}$ and $\{P\}$ states that there are no extra definitions or processes at location φH.

Rule MOVE gives the semantics of migration. A sub-location φH of φ is about to move to a sub-location $\psi H'$ of ψ. On the right hand side, the machine $\Vdash^{\varphi}$ is fully discharged of the location H. Note that P can be executed at any time, whereas Q can only be executed after the migration. Rule MOVE says that migration on the network is based on sub-locations but not objects nor processes. When a migration order is executed, the continuation process moves with all the definitions and processes present at the same sub-location. Nevertheless, we can encode object (or process) migration by defining a fresh sub-location and uniquely attaching an object/process to it. Then the migration of the sub-location will be equivalent to the migration of the object/process.

Names and configuration binding. In the distributed join calculus, every name is defined in at most one local solution; rule MESSAGE-COMM hence applies at most once for every message, delivering the message to a unique location [FG02]. Similarly, we also assume that the rightmost host H_n defines the location φ uniquely. This condition is preserved dynamically thanks to the freshness condition of rule LOC-DEF.

In the semantics, the rule OBJ-DEF (*resp.* LOC-DEF) introduces a fresh variable x (*resp.* H) that is free in the definitions and processes of the whole configuration. But the fact that x (*resp.* H) appears on the left hand side of the machine definition means that the free variable is defined in the configuration. More precisely, for a configuration $\mathcal{C} = (\mathcal{D}_1 \Vdash^{\varphi_i} \mathcal{P}_i)_i$, we say that x is defined in $\mathcal{C}$, noted $\mathcal{C} \vdash x$, when there exists i such that $x.D$ appears in $\mathcal{D}_i$. Similarly, we say that H is defined in $\mathcal{C}$, noted $\mathcal{C} \vdash H$, when there exists i such that $H[\mathcal{D} : \mathcal{P}]$ appears in $\mathcal{D}_i$. This notion of configuration binding will be used in the definition of the semantics of pointcuts in Section 3.2.

3.1.4 A companion example

In the rest of the chapter, we will use a cache replication example. To implement the running example, we assume a dictionary library *dict* with three labels:

- $create(x)$ returns an empty dictionary on $x.getDict$;
- $update(d, k, v, x)$ updates the dictionary d with value v on key k, returning the dictionary on $x.getDict$;
- $lookup(d, k, r)$ returns the value associated to k in d on $r.reply$

We also assume the existence of strings, which will be used for keys of the dictionary, written `"name"`.

The cache we consider is similar to the buffer described in Section 3.1.1 but with a permanent state containing a dictionary and a *getDict* label to receive the (possibly updated) dictionary from the *dict* library:

$$\begin{array}{l}\mathsf{obj}\ c = \ put(k,v)\ \&\ state(d) \triangleright dict.update(d,k,v,c)\\ \qquad\ \mathsf{or}\ \ get(k,r)\ \&\ state(d) \triangleright dict.lookup(d,k,r)\ \&\ c.state(d)\\ \qquad\ \mathsf{or}\ \ getDict(d) \triangleright c.state(d)\\ \mathsf{in}\ \ dict.create(c)\end{array}$$

For the moment, we just consider a single cache and a configuration containing a single machine as follows:

$$\begin{array}{l}c.[put(k,v)\ \&\ state(d) \triangleright dict.update(d,k,v,c),\\ \quad get(k,r)\ \&\ state(d) \triangleright dict.lookup(d,k,r)\ \&\ c.state(d),\\ \quad getDict(d) \triangleright c.state(d)],\\ r.[reply(n) \triangleright out.print(n)]\\ \Vdash^H c.state(d_0)\ \&\ c.get(\texttt{"foo"}, r)\ \&\ c.put(\texttt{"bar"}, 5)\end{array}$$

At this point, two reactions can be performed, involving $c.state(d_0)$ and either $c.get(\texttt{"foo"}, r)$ or $c.put(\texttt{"bar"}, 5)$. Suppose that *put* is (non-deterministically) chosen. The configuration amounts to:

$$Rules \Vdash^H dict.update(d_0, \texttt{"foo"}, 5, c)\ \&\ c.get(\texttt{"foo"}, r)$$

where *Rules* represents the named definitions introduced so far. $c.get(\texttt{"foo"}, r)$ can no longer react, because there are no *c.state* messages in the solution anymore. *dict* passes the updated dictionary d_1, which is passed in the message *c.state* using reaction on label *c.getDict*.

$$Rules \Vdash^H c.state(d_1)\ \&\ c.get(\texttt{"foo"}, r)$$

Now, $c.get(\texttt{"foo"}, r)$ can react with the new message $c.state(d_1)$, yielding:

$$Rules \Vdash^H c.state(d_1)\ \&\ r.reply(5)$$

Finally, 5 is printed out (consuming the *r.reply* message) resulting in the terminal configuration:

$$Rules \Vdash^H c.state(d_1)$$

3.1.5 Bootstrapping distributed communication

Since the join calculus is lexically scoped, programs executed on different machines do not initially share any port name; therefore, they would normally not be able to interact with one another. To bootstrap a distributed computation, it is necessary to exchange a few names, using a name server. The name server NS offers a service to associate a name to a constant string— $NS.register(\texttt{"x"}, x)$—and to look up a name based on a string —$NS.lookup(\texttt{"x"}, r)$, where the value is sent on *r.reply*.

Note that in order to make the definition of processes more readable, we present some part of processes in a functional programming style that can either be encoded in the join calculus, or can already be present in the language (*e.g.*, in JoCaml). In particular, we will use the notions of lists, strings, integers, equality testing, conditionals (`if-then-else`),

$D \stackrel{\text{def}}{=}$	$\ldots$	**Definitions**
	$\langle Pc, Ad \rangle$	pointcut/advice pair
	$\langle Pc, Ad \rangle_\bullet$	activated aspect
$Pc \stackrel{\text{def}}{=}$		**Pointcuts**
	$\mathsf{contains}(x.M)$	reaction pattern binder
	$\mathsf{host}(h)$	location binder
	$\neg Pc$	negation
	$Pc \wedge Pc$	conjunction
	$\mathsf{causedBy}(Pc)$	causality
$Ad \stackrel{\text{def}}{=}$		**Advices**
	P	any process
	$Ad \,\&\, Ad$	parallel composition
	$\mathsf{proceed}$	proceed

Figure 3.4: Syntax of aspects in the aspect join calculus

and a particular variable `lhost` that represents the current location on which a process is executing.

3.2 The aspect join calculus

We now describe the *aspect join calculus*, an extension of the distributed objective join calculus with aspects. Support for crosscutting in a programming language is characterized by its join point model [MKD03]. A join point model includes the description of the points at which aspects can potentially intervene, called *join points*, the means of specifying the join points of interest, here called *pointcuts*, and the means of effecting at join points, called *advices*. We first describe each of these elements in turn, from a syntactic and informal point of view, before giving the formal semantics of aspect weaving in the aspect join calculus. The syntax of aspects is presented in Figure 3.4.

3.2.1 Defining the join point model

Join points. Dynamic join points reflect the steps in the execution of a program. For instance, in AspectJ [KHH+01] join points are method invocations, field accesses, etc. In functional aspect-oriented programming languages, join points are typically function applications [DTK06, WZL03].

The central computational step of any chemical language is the application of a reaction rule, here specified by Rule Red. Therefore, a *reaction join point* in the aspect join calculus is a pair $(\varphi, x.M)$, where φ is the location at which the reduction occurs, and $x.M$ is the matched synchronization pattern of the reduction.

In pointcut definition, it is often of interest to know not only the current reaction join point but also the *causality tree* of reaction join points that gave rise to it. Therefore, we introduce the general notion of *join point (with causality)* to denote a tree of reaction join points. More precisely, a join point (with causality) is

- either an empty tree, noted $\bot$,

$\mathsf{fn}(\mathsf{contains}(x.M))$	$=$	$\mathsf{fn}(x.M)$
$\mathsf{fn}(\mathsf{host}(h))$	$=$	h
$\mathsf{fn}(Pc \wedge Pc')$	$=$	$\mathsf{fn}(Pc) \cup \mathsf{fn}(Pc')$
$\mathsf{fn}(\neg Pc)$	$=$	$\mathsf{fn}(Pc)$
$\mathsf{fn}(\mathsf{causedBy}(Pc))$	$=$	$\mathsf{fn}(Pc)$
$\mathsf{fn}(\mathsf{proceed})$	$=$	$\emptyset$
$\mathsf{fn}(\langle Pc, Ad \rangle)$	$=$	$\mathsf{fn}(Ad) \setminus \mathsf{fn}(Pc)$

Figure 3.5: Definition of free names for aspects

- or a node, noted $jp + \bar{J}$, where jp is a reaction join point and $\bar{J}$ is a list of join points (with causality) whose size is equal to the number of messages in the pattern M.

The idea is that each child represents the causal history of each message in the pattern of the reaction join point node. We note $J' \preceq J$ to indicate that J' is a subtree of J and $J' \prec J$ to indicate that moreover J' is different from J.

Pointcuts. The aspect join calculus includes two basic pointcut designators, *i.e.*, functions that produce pointcuts: contains for reaction rules selection, and host for host selection. The pointcut $\mathsf{contains}(x.M)$ selects any reaction rule that contains the pattern $x.M$ as left hand part, where the variables occurring in $\mathsf{contains}(x.M)$ are bound to the values involved in the reaction join point. In the same way, the pointcut $\mathsf{host}(h)$ binds h to the location of the reaction join point. A pointcut can be also be constructed by negations and conjunctions of other pointcuts. Finally, the pointcut $\mathsf{causedBy}(Pc)$ says that Pc matches for a subtree of the current join point. The semantics of pointcuts is formally described in Section 3.2.3.

The free variables of a pointcut (as defined in Figure 3.5) are bound to the values of the matched join points. In this way, a pointcut acts as a binder of the free variables occurring in the corresponding advice, as standard in aspect-oriented languages. Consider for instance the pointcut $\mathsf{contains}(x.M)$. If x is free, the pointcut will match any reaction whose pattern includes M, irrespective of the involved object, and that object will be bound to the identifier x in the advice body. If x is not a free name, the pointcut will match any reaction *on the object denoted by* x, whose pattern includes M. Note that similarly to synchronization patterns in the join calculus, we require the variables occurring in a pointcut to be linear. This ensures that unions of substitutions used in the definition of a semantics of pointcuts (Fig. 3.6) are always well defined.

In the following, when the variable to be matched is not interesting (in the sense that it is not used in the advice), we use the $*$ notation. For instance, the pointcut $\mathsf{contains}(*.put(k, v))$ matches all reactions containing $put(k, v)$ on any object, without binding the name of the object.

Advices. An advice body Ad is a process to be executed when the associated pointcut matches a join point. This process may contain the special keyword proceed. During the reduction, proceed is substituted by the resulting process P of the matched reaction. Note that contrarily to the common practice in AOP, it is not possible to modify the process P by altering the substitution that is applied to it. This is because the notion of arguments of a reaction is not easy to set up in the join calculus as it should be induced by the substitution and not by the order in which they appear in the reaction join point. Nevertheless, it is still possible to skip using proceed and trigger another process instead. Free names of an advice are defined in Figure 3.5.

Aspects. To introduce aspects in the calculus, we extend the syntax of definitions D with pointcut/advice pairs (Figure 3.4). This means that an object can have both reaction rules and possibly many pointcut/advice pairs. This modeling follows symmetric approaches to pointcut and advice, like CaesarJ [AGMO06] and EScala [GSM+11], where any object has the potential to behave as an aspect. Free names of an aspect are defined in Figure 3.5.

The following example defines an object *replicate* that, when sent a *deploy* message with a given cache replicate object c and a host H', defines a fresh sub-location φH, migrates it to host H', and creates a new replication aspect:

$$
\begin{array}{ll}
\Vdash^{\varphi} & \mathsf{obj}\ \mathit{replicate} = \\
& \quad \mathit{deploy}(c, H') \triangleright \quad H[\mathsf{go}(H'); \mathsf{obj}\ \mathit{rep} = \\
& \qquad\qquad\qquad\qquad \langle \mathsf{contains}(*.\mathit{put}(k,v)) \wedge \mathsf{host}(h), \\
& \qquad\qquad\qquad\qquad\ \ \mathsf{if}\ (H' \notin h)\ \mathsf{then}\ c.\mathit{put}(k,v)\ \&\ \mathsf{proceed} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{proceed}\rangle] \\
& \quad \mathsf{in}\ \mathit{NS.register}(\texttt{"replicate"}, \mathit{replicate})
\end{array}
$$

The advice body replicates on c every *put* message received by a cache object and makes an explicit use of the keyword proceed in order to make sure that the intercepted reaction does occur. The condition $(h \neq H')$ in the advice is used to avoid replication to apply to reactions that happen on a sub-location of the location where the aspect is deployed. Indeed, the aspect must not replicate local modifications of the cache.

3.2.2 Customized reactions

With a single notion of reaction, we are forced to consider a single weaving semantics that applies uniformly to all reactions. In practice, however, exposing each and every join point to aspects can be a source of encapsulation breach as well as a threat to modular reasoning. This issue has raised considerable debate in the AOP community [KM05, Ste06], and several proposal have been made to restrict the freedom enjoyed by aspects (*e.g.*, [BTI14, OSC10, SPAK10, SGR+10]). We now present three variants of weaving semantics.

First of all, it is important for programmers to be able to declare certain reactions as *opaque*, in the sense that they are internal and cannot be woven. This is similar to declaring a method `final` in Java in order to prevent further overriding.

For the many cases in which the semantics of asynchronous event handling is sufficient, it is desirable to be able to specify that aspects can only *observe* a given reaction, meaning that advices are not given the ability to use proceed at all, and are all executed in parallel. This gives programmers the guarantee that the original reaction happens unmodified, and that aspects can only "add" to the resulting computation.

The full aspect join calculus therefore includes three possible weaving semantics, which can be specified per-reaction: opaque ($\blacktriangleright$), observable ($\circledcirc$), and asynchronously advisable ($\triangleright$). The default semantics is asynchronous advisable.

Per-reaction weaving in practice. To illustrate the benefits of different weaving semantics, we refine the definition of a cache object to ensure strong properties with respect to aspect interference as follows:

$$
\begin{array}{l}
\mathsf{obj}\ c = \ \mathit{put}(k,v)\ \&\ \mathit{state}(d) \circledcirc \mathit{dict.update}(d,k,v,c) \\
\qquad \mathsf{or}\ \ \mathit{get}(k,r)\ \&\ \mathit{state}(d) \circledcirc \mathit{dict.lookup}(d,k,r)\ \&\ c.\mathit{state}(d) \\
\qquad \mathsf{or}\ \ \mathit{getDict}(d) \blacktriangleright c.\mathit{state}(d) \\
\mathsf{in}\ \mathit{dict.create}(c)
\end{array}
$$

$$
\begin{array}{rcl}
(\varphi, x'.M') + \bar{J} \Vdash \mathsf{contains}(x.M) & = & \left\{ \begin{array}{l} \tau \text{ minimal substitution s.t.} \\ \quad x\tau = x' \text{ and } M\tau \subseteq M' \\ \\ \bot \text{ otherwise} \end{array} \right. \\
(\varphi, x'.M') + \bar{J} \Vdash \mathsf{host}(h) & = & \{h \mapsto \varphi\} \\
J \Vdash Pc \wedge Pc' & = & J \Vdash Pc \cup J \Vdash Pc' \\
J \Vdash \neg Pc & = & \left\{ \begin{array}{l} \{\ \} \text{ when } J \Vdash Pc = \bot \\ \bot \text{ otherwise} \end{array} \right. \\
J \Vdash \mathsf{causedBy}(Pc) & = & \left\{ \begin{array}{l} J' \Vdash Pc \text{ for some } J' \text{ s.t.} \\ \quad J' \prec J \text{ and } J' \Vdash Pc \neq \bot \\ \bot \text{ otherwise} \end{array} \right.
\end{array}
$$

Figure 3.6: Semantics of pointcuts

Reactions on both *put* and *get* are declared *observable*, in order to ensure that aspects cannot prevent them from occurring normally. Additionally, reactions on the internal *getDict* channel are *opaque* to enforce strong encapsulation.

3.2.3 Semantics

Semantics of pointcuts. The matching relation, noted $jp \Vdash Pc$, returns either a substitution τ from free names of Pc to names or values of jp, or a special value $\bot$ meaning that the pointcut does not match. That is, we enriched the notion of boolean values to a richer structure (here substitutions), as commonly done in aspect-oriented programming languages in particular. We note $\{\ \}$ the empty substitution, and consider it as the canonical true value. We note $\cup$ the join operation on disjoint substitutions that returns $\bot$ as soon as one of the substitution is $\bot$. Note that conjunction pointcuts are defined only on substitutions that are disjoint, but because variables occur linearly in pointcuts, we have the guarantee that this is always the case. The matching relation is defined by induction on the structure of the pointcut in Figure 3.6. We say that a pointcut Pc *does not match* a join point J when $J \Vdash Pc = \bot$.

In the rule for the $\mathsf{contains}(x.M)$ pointcut, the inclusion of patterns $M\tau \subseteq M'$ is defined as the inclusion of the induced multiset of messages. This implies in particular that when a label occurs more than once in the messages of the matched pattern, there may be several substitutions that induce the inclusion of patterns. This means that the semantics of pointcuts, as the reduction of the join calculus, is not deterministic.

For instance, suppose that the cache replication aspect defined previously has been deployed and that the emitted join point is:

$$(\varphi, x.put(\texttt{"bar"}, 5)\ \&\ state(\mathsf{d}))\ +\ \bar{J}$$

Then, the pointcut of the aspect:

$$\mathsf{contains}(*.put(k, v)) \wedge \mathsf{host}(h)$$

matches, with partial bijection:

$$\tau = \{k \mapsto \texttt{"bar"}, v \mapsto 5, h \mapsto \varphi\}$$

$$
\begin{array}{lcl}
\{0\}_J & = & 0 \\
\{x.M\}_J & = & x.\{M\}_J \\
\{\mathsf{obj}\ x = D\ \mathsf{in}\ P\}_J & = & \mathsf{obj}\ x = D\ \mathsf{in}\{P\}_J \\
\{\mathsf{go}(H); P\}_J & = & \mathsf{go}(H); \{P\}_J \\
\{H[P]\}_J & = & H[\{P\}_J] \\
\{P\ \&\ P\}_J & = & \{P\}_J\ \&\ \{P\}_J \\
\{l(\bar{v})\}_J & = & l_{\mathbf{J}}(\bar{v}) \\
\{M\ \&\ M\}_J & = & \{M\}_J\ \&\ \{M\}_J
\end{array}
$$

Figure 3.7: Tagging of causal history.

Note that the variable d is not mapped by τ because it is not captured by the pointcut.

The rule for the $\mathsf{host}(h)$ pointcut always returns the substitution that associates h with the location of the matched pattern. The semantics of the negation and conjunction is an extension of the traditional boolean semantics to truth values that are substitutions. The rule for the $\mathsf{causedBy}(Pc)$ pointcut returns the substitution that matches Pc for any strict sub join point J' of J, that is any join point in the causal history of J. It returns $\perp$ when no join point matches Pc.

If a cache replication aspect is deployed on each host of interest, then aspects will indefinitely replicate the cache replicated by aspects on other hosts. In AWED [BNSV+06], this livelock is prevented by excluding join points produced within the body of the aspect, using the `within` pointcut designator. Similarly, it is common in aspect languages to use control-flow related pointcuts in order to be able to discriminate join points caused by others (*e.g.*, `cflow` and `cflowbelow` in AspectJ [KHH+01]). Most distributed aspect languages and systems also support distributed control flow, although in a synchronous setting. This indicates that a notion of causality is required in order to express this kind of pointcuts. Let us illustrate with the cache replication example. To be able to identify aspect-specific activity, we declare an aspect object, with a specific label *rput* whose goal is to make the activity of the aspect visible. Then the new definition of the cache replication aspect below also excludes the activity *caused by* a cache replication aspect using the pointcut $\neg\,\mathsf{causedBy}(*.rput)$.

$$
\begin{array}{l}
\Vdash^{\varphi}\ \ \mathsf{obj}\ replicate = \\
\quad deploy(c, H') \triangleright\ \ H[\mathsf{go}(H'); \\
\qquad\qquad \mathsf{obj}\ rep = \\
\qquad\qquad\quad rput(k, v) \triangleright c.put(k, v) \\
\qquad\qquad\ \ \mathsf{or}\ \langle \mathsf{contains}(*.put(k, v)) \wedge \mathsf{host}(h) \wedge \\
\qquad\qquad\quad \neg\,\mathsf{causedBy}(\mathsf{contains}(*.rput(*, *))), \\
\qquad\qquad\ \ \mathsf{if}\ (h \neq H')\ \mathsf{then}\ c.put(k, v)\ \&\ \mathsf{proceed} \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{proceed}\rangle] \\
\quad \mathsf{in}\ \ NS.register(\texttt{"replicate"}, replicate)
\end{array}
$$

This ensures that a cache replication aspect never matches a *put* join point that has been produced by the rule $rput(k, v) \triangleright c.put(k, v)$, thereby ignoring aspect-related computation.

Remembering causality in processes. In order to conserve and propagate the causal history during the reduction, each message $l(\bar{v})$ are tagged with the join point J that causes it, noted $l_{\mathbf{J}}(\bar{v})$. Given a pattern M that is matched during the reduction, we note $\{M\}_{\bar{J}}$ the pattern tagged with the causal history of each message present in the pattern (note that M and $\bar{J}$ have to be of the same size) as defined by:

- $\{l(\bar{v})\}_{[J]} = l_{\mathbf{J}}(\bar{v})$

DEPLOY
$x.[\langle Pc, Ad\rangle] \Vdash^{\varphi} \longrightarrow x.[\langle Pc, Ad\rangle_{\bullet}] \Vdash^{\varphi}$

RED/NOASP
$x.[M \triangleright P] \Vdash^{\varphi} x.\{M\sigma\}_{\bar{J}} \longrightarrow x.[M \triangleright P] \Vdash^{\varphi} \{P\sigma\}_{J'}$

when no pointcut of an activated aspect matches J'.

RED/ASP
$x.[M \triangleright P] \Vdash^{\varphi} x.\{M\sigma\}_{\bar{J}} \quad \|_{i\in I} \quad x_i.[\langle Pc_i, Ad_i\rangle_{\bullet}] \Vdash^{\psi_i} \longrightarrow$
$x.[M \triangleright P] \Vdash^{\varphi} \quad \|_{i\in I} \quad x_i.[\langle Pc_i, Ad_i\rangle_{\bullet}] \Vdash^{\psi_i} Ad_i[\{P\sigma\}_{J'}/\,\mathsf{proceed}]\tau_i$

where $J' \Vdash Pc_i = \tau_i$ for all $i \in I$
and no other activated aspect matches J'.

RED/OPAQUE
$x.[M \blacktriangleright P] \Vdash^{\varphi} x.\{M\sigma\}_{\bar{J}} \longrightarrow x.[M \blacktriangleright P] \Vdash^{\varphi} \{P\sigma\}_{J'}$

RED/OBSERVABLE
$x.[M \circledcirc P] \Vdash^{\varphi} x.\{M\sigma\}_{\bar{J}} \quad \|_{i\in I} \quad x_i.[\langle Pc_i, Ad_i\rangle_{\bullet}] \Vdash^{\psi_i} \longrightarrow$
$x.[M \circledcirc P] \Vdash^{\varphi} \{P\sigma\}_{J'} \; \|_{i\in I} \; x_i.[\langle Pc_i, Ad_i\rangle_{\bullet}] \Vdash^{\psi_i} \{Ad_i\tau_i\}_{J'}$

where $J' \Vdash Pc_i = \tau_i$ for all $i \in I$
and no other activated aspect matches J'.

In every reduction rule, J' stands for $(\varphi, x.M\sigma) + \bar{J}$.

Figure 3.8: Semantics of aspect weaving

- $\{M \,\&\, M'\}_{\bar{J} \,@\, \bar{J'}} = \{M\}_{\bar{J}} \,\&\, \{M'\}_{\bar{J'}}$

Figure 3.7 presents tagging for processes that are produced by a reduction. Here, the idea is to tag each message that has been produced by a reduction.

Initially, a process P is started with an empty causal history, $\{P\}_{\perp}$.

Semantics of aspect weaving. Figure 3.8 presents the semantics of aspect weaving. All rules of Figure 3.3 are preserved, except for Rule RED because this is where weaving takes place. This rules is split into four rules, all of which depend on currently activated aspects as expressed by the following rule.

Rule DEPLOY corresponds to the asynchronous deployment of a pointcut/advice pair $x.[\langle Pc, Adv\rangle]$ by marking the pair as activated $x.[\langle Pc, Adv\rangle_{\bullet}]$. Note that activated pairs are not directly user-definable. The presence of this rule is crucial in the semantics because it allows to activate aspects one by one asynchronously. Another possible semantics would have been to deploy synchronously altogether pointcut/advice pairs of the same definition, but then it would have caused extra synchronization in the translation to the core join calculus (see Sect. 3.3).

Rule RED/NOASP is a direct reminiscence of Rule RED in case where no activated pointcut matches. Note that the new causal history is propagated to the produced process $P\sigma$.

Rule RED/ASP defines the modification of Rule RED in presence of aspects. If there is an aspect x_i with an activated pointcut/advice pair $x_i.[\langle Pc, Ad\rangle_{\bullet}]$ such that Pc matches

the join point with substitution τ, the advice Ad is executed with the process P substituting the keyword proceed and where the variables bound by the pointcut are substituted according to τ. The side condition of Rule Red/Asp is that all Pc_is are the activated pointcuts that match the current join point $(\varphi, x.M\sigma)$. In particular, when two pointcut/advice pairs of the same object definition match, we can have $x_i = x_j$ and $\psi_i = \psi_j$. Note that all advices associated to a pointcut that matches are executed in parallel.

Rule Red/Opaque is computationally the same as Rule Red/NoAsp, since activated aspects are essentially ignored when an opaque reaction occurs.

Rule Red/Observable proceeds the original reaction in parallel with the application of all deployed pointcut/advice pairs that match the join point. Note that in this rule, an advice has to be a simple process, and hence cannot use proceed (this can be guaranteed by a simple type system).

Coming back to the cache example, the synchronization pattern reacts to become:

$$\begin{array}{ll} & x.put(\texttt{"bar"}, 5) \,\&\, state(\mathsf{d}) \\ \longrightarrow & c.put(\texttt{"bar"}, 5) \,\&\, dict.update(\mathsf{d}, \texttt{"bar"}, 5, x) \end{array}$$

The original operation on *dict* to update d is performed, in addition to the replication on c.

3.2.4 Why objects?

When designing the aspect join calculus, we considered defining it on top of the standard join calculus with explicit distribution, but without objects. However, it turns out that doing so would make the definition of aspects really awkward and hardly useful. Consider the standard join calculus definition of a buffer producer (adapted from [FG96b]):

$$\begin{array}{l} \mathsf{def}\ \mathit{make\_buffer}(k) \triangleright \\ \quad \mathsf{def}\quad \mathit{put}(n) \mid \mathit{empty}() \triangleright \mathit{some}(n) \\ \quad\ \wedge\quad \mathit{get}(r) \mid \mathit{some}(n) \triangleright r(n) \mid \mathit{empty}() \\ \mathsf{in}\ \mathit{empty}() \mid k(\mathit{get}, \mathit{put}) \end{array}$$

make_buffer takes as argument a response channel k on which the two newly-created channels *get* and *put* are passed (hence representing the new buffer). Crucially, the channel names *get* and *put* are local and not meaningful *per se*; when the definitions are processed, they are actually renamed to fresh names (rule str-def in [FG96b]). Therefore, there is no way for an aspect to refer to "a reaction that includes a message on the *get* channel". Doing so would require modifying *make_buffer* to explicitly pass the newly-created channels also to the aspect, each time it is executed. An aspect would then have to match on all reactions and check if the involved channels include one of the ones it has been sent. In addition, the explicit modification of *make_buffer* defeats the main purpose of aspects, which is separation of concerns. A *make_buffer* that explicitly communicates its created channels to a replication aspect is not a general-purpose entity that can be reused in different contexts (*e.g.*, without replication).

The objective join calculus, on the other hand, includes both object names and *labels*. Conversely to object names, labels have no local scope and are not subject to renaming [FLMR03]. They constitute a "shared knowledge base" in the system, which aspects can exploit to make useful quantification. This is similar to how method names are used in the pointcuts of object-based aspect-oriented languages.

The argumentation above also explains why we have chosen not to include classes as in [FLMR03] in our presentation of the aspect join calculus. Classes support extensible definitions, but do not contribute anything essential with respect to naming and

quantification.

3.3 From the aspect join calculus to the join calculus

In this section, we present a translation of the aspect join calculus into the core join calculus. This allows us to specify an implementation of the weaving algorithm, and to prove it correct via a bisimilarity argument. The translation is used in Section 3.4 to implement Aspect JoCaml on top of JoCaml [FLFMS03], an implementation of the join calculus in OCaml.

3.3.1 General approach

The translation approach consists is considering that an aspect is a standard object that receives messages from the weaver to execute a particular method that represents its advice. This is the usual way to compile aspects to a target object-oriented language without aspects [HH04].

Aspect weaving. In order to determine whether an aspect applies or not, the translation must account for aspect weaving. Note that the description of the semantics of the aspect join calculus leaves open the question of the underlying aspect weaving infrastructure. The naive approach consists in relying on a central weaver that coordinates all distributed computation and triggers the weaving of all aspects. This centralized approach, described in [Tab10], is however not realistic in a distributed setting and is therefore not useful as an implementation strategy.

Decentralized weaving. The first step towards a flexible decentralized weaving architecture is to distribute weavers across several hosts. Having several weavers raises the question of their granularity. Because each reaction can have a specific weaving semantics (Section 3.2.2), we consider one weaver per reaction, in charge of performing aspect weaving for a given reaction.

In order to mediate between aspects and weavers, we introduce the notion of a *weaving registry*, in direct analogy with, for instance, registries in Java RMI. A weaving registry w_R is in charge of bootstrapping the binding between weavers and aspects. The definitions D_{w_R} available to communicate with the weaving registry are given by:

$$\begin{aligned} D_{w_R} = \quad & get_{asp}(k) \,\&\, asp_{act}(\bar{a}) \triangleright k(\bar{a}) \,\&\, asp_{act}(\bar{a}) \\ & deploy(a) \,\&\, asp_{act}(\bar{a}) \triangleright asp_{act}(a\,, \bar{a}) \end{aligned}$$

Aspects get activated by registering to the weaving registry through the channel *deploy* and local weavers get the current list of activated aspects each time a reaction is triggered by passing a continuation to the channel get_{asp}.[6] The weaving registry is executing at location H_W and is known by all other processes.

[6] It is important that weavers ask for the current list of activated aspects *before* weaving a reaction because it guarantees the consistency of the knowledge of the list of activated aspects between distributed local weavers. Indeed, a more asynchronous and decentralized translation where newly activated aspects are broadcast to local weavers—which are then responsible for storing the list of activated aspects—would not be equivalent to the semantics of Figure 3.8. This is because the asynchronous broadcast of aspects would allow a weaver to use a list of activated aspects that is inconsistent with the list of activated aspects known by other weavers. But the rules of Figure 3.8 assume a global consistent view of the list of activated aspects.

Processes

$$
\begin{array}{rcl}
[\![0]\!]_J & = & 0 \\
[\![x.M]\!]_J & = & x.[\![M]\!]_J \\
[\![\mathsf{obj}\ x = D\ \mathsf{in}\ P]\!]_J & = & \mathsf{obj}\ \mathcal{W}(D_r, x)\ \mathsf{in} \\
 & & \mathsf{obj}\ x = [\![D_r]\!]_x\ \mathsf{in}\ [\![D_a]\!]\ \&\ [\![P]\!]_J \\
[\![\mathsf{go}(H); P]\!]_J & = & \mathsf{go}(H); [\![P]\!]_J \\
[\![H[P]]\!]_J & = & H[[\![P]\!]_J] \\
[\![P\ \&\ P']\!]_J & = & [\![P]\!]_J\ \&\ [\![P']\!]_J
\end{array}
$$

Definitions

$$
\begin{array}{rcl}
[\![M \triangleright P]\!]_x & = & [\![M]\!]_{\bar{J}_M}\ \mathsf{obj}\ ret = proceed(J) \triangleright [\![P]\!]_J \\
 & & \quad \mathsf{in}\ \mathsf{let}\ J = (\texttt{lhost}, x.M) + \bar{J}_M \\
 & & \quad\quad \mathsf{in}\ W_{x.M}.weave(ret.proceed, J) \\
[\![M \circledcirc P]\!]_x & = & [\![M \triangleright P]\!]_x \\
[\![M \blacktriangleright P]\!]_x & = & [\![M]\!]_{\bar{J}_M} \triangleright \mathsf{let}\ J = (\texttt{lhost}, x.M) + \bar{J}_M \\
 & & \quad\quad \mathsf{in}\ [\![P]\!]_J \\
[\![D\ \mathsf{or}\ D']\!]_x & = & [\![D]\!]_x\ \mathsf{or}\ [\![D']\!]_x
\end{array}
$$

where $\bar{J}_M = J_1, .., J_n$ with $n = |M|$

Messages

$$
\begin{array}{rcl}
[\![l(\bar{v})]\!]_J & = & l(\bar{v}, J) \\
[\![l_{\mathbf{J}}(\bar{v})]\!]_J & = & l(\bar{v}, \mathbf{J}) \\
[\![M\ \&\ M']\!]_J & = & [\![M]\!]_J\ \&\ [\![M']\!]_J \\
 & & \\
[\![l(\bar{v})\ \&\ M']\!]_{J,\bar{J}} & = & l(\bar{v}, J)\ \&\ [\![M']\!]_{\bar{J}}
\end{array}
$$

Pointcuts

$$
\begin{array}{rcl}
[\![\mathsf{contains}(x.M)]\!] & = & \lambda(\varphi, x'.M') + \bar{J}. \\
 & & \quad \mathsf{if}\ M\tau \subseteq M'\ \mathsf{then}\ [x', \tau]\ \mathsf{else}\ [] \\
[\![\mathsf{host}(H)]\!] & = & \lambda(\varphi, x'.M') + \bar{J}.\ [[\varphi]] \\
[\![\neg Pc]\!] & = & \lambda J.\ \mathsf{if}\ [\![Pc]\!]\ J = []\ \mathsf{then}\ [[]]\ \mathsf{else}\ [] \\
[\![Pc \wedge Pc']\!] & = & \lambda J.\ \mathsf{shuffle}([\![Pc]\!]\ J, [\![Pc']\!]\ J) \\
[\![\mathsf{causedBy}(Pc)]\!] & = & \lambda jp + \bar{J}.\ \mathsf{concat}(\mathsf{map}\ [\![Pc]\!]_{rec}\ \bar{J}) \\
 & & \\
[\![Pc]\!]_{rec} & = & \lambda\bot.\ [] \\
[\![Pc]\!]_{rec} & = & \lambda jp + \bar{J}.\ [\![Pc]\!]\ (jp + \bar{J})\ @ \\
 & & \quad\quad \mathsf{concat}(\mathsf{map}\ [\![Pc]\!]_{rec}\ \bar{J})
\end{array}
$$

Aspects

$$
\begin{array}{rcl}
[\![\mathsf{proceed}]\!]_J & = & proceed(J) \\
[\![\langle Pc, Ad\rangle]\!] & = & \mathsf{obj}\ adv = advice(proceed, J, \bar{v}_{Pc}) \triangleright \\
 & & \quad [\![Ad]\!]_J \\
 & & \mathsf{in}\ w_R.deploy([\![Pc]\!], adv.advice)
\end{array}
$$

where $\bar{v}_{Pc}$ is the list of variables occurring free in Pc.

Figure 3.9: Translating the aspect join calculus to the join calculus

Translation overview. Therefore, given an aspect join calculus configuration:

$$\mathcal{C} = \mathcal{D}_1 \Vdash^{\varphi_1} \mathcal{P}_1 \quad \| \quad \cdots \quad \| \quad \mathcal{D}_n \Vdash^{\varphi_n} \mathcal{P}_n$$

we construct a distributed join calculus configuration without aspects by translating definitions, processes and aspects—following the specific translation described below in Section 3.3.2—and introducing a weaving registry w_R that monitors the list of activated aspects:

$$\begin{array}{rcl} [\![\mathcal{C}]\!] & = & [\![\mathcal{D}_1]\!] \Vdash^{\varphi_1} [\![\mathcal{P}_1]\!] \quad \| \cdots \| \quad [\![\mathcal{D}_n]\!] \Vdash^{\varphi_n} [\![\mathcal{P}_n]\!] \\ & & \| \quad w_R.D_{w_R} \Vdash^{H_w} w_R.asp_{act}([\![\bar{a}]\!]) \end{array}$$

where $\bar{a}$ is the list of activated aspects in $\mathcal{D}_1, \ldots, \mathcal{D}_n$. The idea is to manipulate an explicit join point in every reaction rule. This join point triggers a protocol with the weaver to decide whether or not some aspect intercepts the reaction rule and must be executed.

3.3.2 Translation

The translation of processes, definitions and aspects, from the aspect join calculus to the standard join calculus, is defined in Figure 3.9.

Processes. The rules for processes recursively propagate the translation in sub-processes and definitions. The translation of objects requires to distinguish between reaction rules (D_r) and pointcut/advice pairs (D_a) in the original definition D, because each pointcut/advice is translated as a normal object. The translation of reaction rules is done in two steps. First, a local weaver is created for each reaction, using $\mathcal{W}(D_r, x)$, and then each reaction is replaced by a reaction that communicates with its weaver.

Definitions. A reaction rule $M \triangleright P$ is replaced by a call to the weaver, passing a locally-created single-use channel *ret.proceed* to perform the original computation P and the current join point, obtained by collecting join points of the matched pattern $\bar{J}_M$ and adding the current reaction join point $(\texttt{lhost}, x.M)$. It is important that the new channel is locally-created and of a single use because it guarantees that different calls to proceed cannot be interleaved. Note that opaque reactions are not woven and the difference between standard and observable reactions only shows up in the definition of the weaver (Figure 3.10).

Messages. Similarly to the tagging mechanism of messages described in Section 3.2.3, there are three ways to translate a message: (1) if it occurs on the left hand side of a definition, the list of variables is extended with a variable J_i that captures the causal history of the message, (2) if it occurs on the right hand side of a definition, the current join point J is added to the message, (3) if it is a tagged message $l_{\mathbf{J}}$, the tagged join point $\mathbf{J}$ is added to the message.

Pointcuts. Pointcuts are recursively transformed into functions that operate on join points and return a list of substitutions (encoded as lists) that correspond to all the possible matches. When the list is empty, it means that the pointcut does not match. This is the usual folklore way of computing altogether the possible results of a non deterministic function in one step [Wad85]. To preserve the non-deterministic nature of pointcut matching, only one substitution will be chosen randomly by the weaver.

In the definition of $[\![\mathsf{contains}(x.M)]\!]$, the substitution τ is seen as the list, and contains only free variables of M. The definition of $[\![Pc \wedge Pc']\!]$ uses the function $\mathsf{shuffle}$ that takes

two lists of lists and returns the list of lists of all possible concatenations, which can be written using list comprehension as:

$$\mathsf{shuffle}\ ls\ ls' = [\, l \mathbin{@} l' \mid l \leftarrow ls, l' \leftarrow ls' \,].$$

The definition of $[\![\mathsf{causedBy}(Pc)]\!]$ is given by computing every possible match of Pc on every sub join point. concat and map are the usual operations on lists.

Aspects. A pointcut/advice pair $\langle Pc, Ad \rangle$ is translated as an object that holds the advice and is registered with the weaver. The advice has only one channel *advice* which expects the proceed channel, the current join point and the list of variables that are free in the pointcut Pc.

The initialization sends the pointcut/advice pair to the weaving registry W_r by using the dedicated label *deploy*. Note that the pointcut is sent to the weaver but is not checked explicitly in the aspect. Indeed, it is the responsibility of the weaver to decide whether the advice must be executed or not. This is because the weaver must have the global knowledge of which pointcuts match, to perform Rule Red/NoAsp.

Finally, the translation of proceed is obtained by adding the current join point J as argument to *proceed*.

Per-reaction weavers. The per-reactions weavers are defined altogether at the beginning of the translation of an object using the inductive definition $\mathcal{W}(D, x)$, given in Figure 3.10. The idea is to define, for each reaction, a weaver as an object with a *weave* method, used to trigger weaving at a join point. The weaver of an opaque reaction is just the null object.

The definition of the weaver depends on the kind of reaction, observable ($D_{\circledcirc}$) or advisable ($D_{\triangleright}$). In both cases, when the weaver receives $weave(proceed, J)$, it creates a new object W_{init} that defines a fresh channel $aspL()$ whose aim is to get the current list of activated aspect *asp* from the weaving registry by spawning $get_{asp}(W_{init}.aspL)$. When the list is received, the weaver tests the emptiness of the list *ads* of advices whose pointcut matches J (defined using the usual filter functions on lists).

For observable reactions, the weaver $D_{\circledcirc}$ just spawns in parallel a call to proceed and call to advices (using the iter function on lists) for which the pointcut has matched, *by selecting randomly* a substitution from the list of substitutions $\bar{\bar{v}}$ using the non-deterministic selection function select. This corresponds to Rule Red/Obs.
For advisable reactions, the weaver $D_{\triangleright}$ needs to distinguish between two cases, corresponding to the two Rules Red/NoAsp and Red/Asp. If no aspect applies, the weaver executes the original process by sending the message $proceed(J)$ to resume the original computation; this corresponds to Rule Red/NoAsp. Otherwise, the weaver executes all advices present in *ads* asynchronously using again a randomly chosen substitutions from the list $\bar{\bar{v}}$ for each advice. This corresponds to Rule Red/Asp.

Named Definitions. When going to the left-hand side of a configuration, definitions are named by the created object, but the original definition of the object disappears. To conserve the definition of each per-reaction weaver attached to a reaction, a named reaction must be translated to the named translation of the reaction, plus the definition of his corresponding weaver (Figure 3.11). For pointcut/advice pair, the translation is obtained by applying the structural rule Obj-Def to the advising object translation.

The translation of a activated pointcut/advice pair is identical, except for the implicit assumption that the initial message $w_R.deploy([\![Pc]\!], adv.advice)$ has been consumed by

$$
\begin{array}{rcl}
\mathcal{W}(M \triangleright P, x) & = & W_{x.M} = D_{\triangleright} \\
\mathcal{W}(M \circledcirc P, x) & = & W_{x.M} = D_{\circledcirc} \\
\mathcal{W}(M \blacktriangleright P, x) & = & W_{x.M} = 0 \\
\mathcal{W}(D \,\mathsf{or}\, D', x) & = & \mathcal{W}(D, x) \;\mathsf{in}\; \mathsf{obj}\; \mathcal{W}(D', x)
\end{array}
$$

$$
\begin{array}{rl}
D_{\circledcirc} = & \mathit{weave}(\mathit{proceed}, J)\triangleright \\
& \mathsf{obj}\; W_{init} = \mathit{aspL}(\mathit{asps})\triangleright \\
& \quad \mathsf{let}\; \mathit{ads} = \mathsf{filter}\; (\lambda(\bar{\bar{v}}, \_).\bar{\bar{v}} \neq []) \\
& \qquad\qquad \mathsf{map}\; (\lambda(Pc, \mathit{adv}).([\![Pc]\!]\; J, \mathit{adv}))\; \mathit{asps} \\
& \quad \mathsf{in}\; \mathit{proceed}(J)\; \&\; \mathsf{iter}\; (\lambda(\bar{\bar{v}}, \mathit{adv}).\; \mathit{adv}(J, \mathsf{select}(\bar{\bar{v}})))\; \mathit{ads} \\
& \mathsf{in}\; \mathit{get}_{\mathit{asp}}(W_{\mathit{init}}.\mathit{aspL})
\end{array}
$$

$$
\begin{array}{rl}
D_{\triangleright} = & \mathit{weave}(\mathit{proceed}, J)\triangleright \\
& \mathsf{obj}\; W_{init} = \mathit{aspL}(\mathit{asps})\triangleright \\
& \quad \mathsf{let}\; \mathit{ads} = \mathsf{filter}\; (\lambda(\bar{\bar{v}}, \_).\bar{\bar{v}} \neq []) \\
& \qquad\qquad \mathsf{map}\; (\lambda(Pc, \mathit{adv}).([\![Pc]\!]\; J, \mathit{adv}))\; \mathit{asps} \\
& \quad \mathsf{in}\; \mathsf{if}\; \mathit{ads} = [] \\
& \qquad \mathsf{then}\; \mathit{proceed}(J) \\
& \qquad \mathsf{else}\;\; \mathsf{iter}\; (\lambda(\bar{\bar{v}}, \mathit{adv}).\; \mathit{adv}(\mathit{proceed}, J, \mathsf{select}(\bar{\bar{v}})))\; \mathit{ads} \\
& \mathsf{in}\; \mathit{get}_{\mathit{asp}}(W_{\mathit{init}}.\mathit{aspL})
\end{array}
$$

Figure 3.10: Per-reaction weaving

$$
\begin{array}{rcl}
[\![x.[M \triangleright P]]\!] & = & x.[[\![M \triangleright P]\!]_x] \,\mathsf{or}\, W_{x.M}.D_{\triangleright} \\
[\![x.[M \circledcirc P]]\!] & = & x.[[\![M \circledcirc P]\!]_x] \,\mathsf{or}\, W_{x.M}.D_{\circledcirc} \\
[\![x.[M \blacktriangleright P]]\!] & = & x.[[\![M \blacktriangleright P]\!]_x] \\
[\![x.\langle Pc, Ad\rangle]\!] & = & x.[\mathit{advice}(\mathit{proceed}, J, \bar{v}_{Pc}) \triangleright [\![Ad]\!]_J] \\
[\![x.D \,\mathsf{or}\, D']\!] & = & [\![x.D]\!] \,\mathsf{or}\, [\![x.D']\!]
\end{array}
$$

where $\bar{v}_{Pc}$ is the list of variables occurring free in Pc.

Figure 3.11: Translation of named definitions

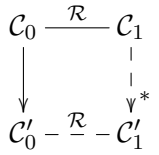
weaving registry. The translation of the list of activated aspects of $w_R.\mathit{asp}_{\mathit{act}}([\![\bar{a}]\!])$ is given by

$$[\![x.\langle Pc, Ad\rangle_{\bullet}\,,\, \bar{a}]\!] = ([\![Pc]\!], x.\mathit{advice})\,,\, [\![\bar{A}]\!]$$

3.3.3 Bisimulation between an aspect join calculus process and its translation

The main interest of translating the aspect join calculus into the core join calculus is that it provides a direct implementation of the weaving algorithm that can be proved to be correct. As usual in concurrent programming languages, the correctness of the algorithm is given by a proof of bisimilarity. Namely, we prove that the original configuration with aspects is bisimilar to the translated configuration that has no aspect. The idea of bisimilarity is to express that, at any stage of reduction, both configurations can perform the same actions in the future. More formally, in our setting, a simulation $\mathcal{R}$ is a relation between configurations such that when $\mathcal{C}_0\ \mathcal{R}\ \mathcal{C}_1$ and $\mathcal{C}_0$ reduces in one step to $\mathcal{C}'_0$, there exists $\mathcal{C}'_1$ such that $\mathcal{C}'_0\ \mathcal{R}\ \mathcal{C}'_1$ and $\mathcal{C}_1$ reduces (in 0, 1 or more steps) to $\mathcal{C}'_1$. We illustrate

this with the following diagram:



A bisimulation is a simulation whose inverse is also a simulation.

To relate a configuration $\mathcal{C}$ with its translation $[\![\mathcal{C}]\!]$, we need to tackle two difficulties:

1. During the evolution of $[\![\mathcal{C}]\!]$, auxiliary messages that have no correspondents in $\mathcal{C}$ are sent for communication between processes, weaver, aspects and the weaving registry.

2. In the execution of $\mathcal{C}$, proceed is substituted by the process P to be executed, whereas in $[\![\mathcal{C}]\!]$, P is executed through a communication with the object where the reaction has been intercepted.

To see the auxiliary communication as part of a reduction rule of the aspect join calculus, we define a notion of standard form for the translated configurations. Let

$$\mathbb{T} = \{\mathcal{C} \mid \exists \mathcal{C}_0, [\![\mathcal{C}_0]\!] \longrightarrow^* \mathcal{C}\}$$

be the set of configurations that come from a translated configuration. We construct a rewriting system $\longrightarrow_{\mathbb{T}}$ for $\mathbb{T}$, based on the reduction rule of the join calculus. Namely, we take Rule Red restricted to the case where the pattern contains either of the dedicated labels: *weave*, *proceed*, *advice*, get_{asp} and *aspL* (the label *deploy* is treated differently as it corresponds to the application of Rule Deploy). In $\mathbb{T}$, those labels only interact alone, or one-by-one with the constant label asp_{act}. So the order in which reaction rules are selected has no influence on the synchronized pattern; in other words, the rewriting system $\longrightarrow_{\mathbb{T}}$ is confluent. Furthermore, it is not difficult to check that this rewriting system is also terminating. Therefore, it makes sense to talk about the normal form of $\mathcal{C} \in \mathbb{T}$, noted $\widetilde{\mathcal{C}}$.

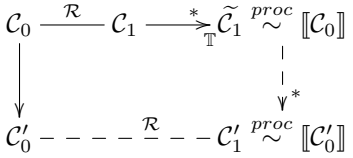
We note $\mathcal{C} \overset{proc}{\sim} \mathcal{C}'$ when $\mathcal{C}'$ is equal to $\mathcal{C}$ where every message $proceed(J)$ is substituted by the process $[\![P]\!]$ to which it corresponds.

Theorem 45

The relation $\mathcal{R} = \{(\mathcal{C}_0, \mathcal{C}_1) \mid [\![\mathcal{C}_0]\!] \overset{proc}{\sim} \widetilde{\mathcal{C}_1}\}$ is a bisimulation. In particular, any configuration is bisimilar to its translation.

The crux of the proof lies in the confluence of $\longrightarrow_{\mathbb{T}}$ which means that once the message $weave(k, jp)$ is sent to the weaver, the translation introduces no further choice in the configuration. That is, every possible choice in $[\![\mathcal{C}]\!]$ corresponds directly to the choice of a reduction rule in $\mathcal{C}$.

Proof: The fact that $\mathcal{R}$ is a simulation just says that the communication between aspects, processes and the weaver simulates the abstract semantics of aspects. More precisely, we show that for any reduction $\mathcal{C}_0 \longrightarrow \mathcal{C}'_0$ using Rule Deploy, Red/Asp or Red/NoAsp, one can find a corresponding reduction chains from $[\![\mathcal{C}_0]\!]$ to $[\![\mathcal{C}'_0]\!]$:



Rule DEPLOY**.**

The activation of an aspect $x.[\langle Pc, Ad\rangle]$ is in one-to-one correspondence with the consumption of the message of the message $w_R.deploy([\![Pc]\!], x.advice)$ by the weaving registry. Note at this point, that the fact that aspect activation is asynchronously described by Rule DEPLOY in the semantics is crucial in the proof.

Rule RED/NOASP**.**

Straightforward.

Rule RED/ASP**.**

Consider the reduction

$$x.l(\bar{v}) \quad \longrightarrow \quad Ad_1[P/\,\mathsf{proceed}] \,\&\, \cdots \,\&\, Ad_n[P/\,\mathsf{proceed}]$$

This rule is simulated by the chain:

$$\begin{array}{lll} x.l(\bar{v}) & \longrightarrow & w.weave(k, jp) \\ & \longrightarrow & Ad_1.advice(k, jp) \,\&\, \cdots \,\&\, Ad_n.advice(k, jp) \\ & \longrightarrow & [\![Ad_1]\!]_{x_1} \,\&\, \cdots \,\&\, [\![Ad_n]\!]_{x_n} \end{array}$$

leading the normal form $\mathcal{C}_1' \overset{proc}{\sim} [\![\mathcal{C}_0']\!]$. Rule DEPLOY is simulated by the first reaction rule in the definition of the weaver.

The converse direction is more interesting as it says that any reduction in the translated configuration can be seen as a step in the simulated reduction of Rule DEPLOY, RED/ASP or RED/NOASP of the original configuration. More precisely, we have to show that any reduction $\mathcal{C}_1 \longrightarrow \mathcal{C}_1'$ can be seen as a reduction between their normal forms. This expressed by the following diagram:

$$\begin{array}{ccccc} \mathcal{C}_0 & \overset{\mathcal{R}}{\text{---}} & \mathcal{C}_1 & \overset{*}{\underset{\mathbb{T}}{\longrightarrow}} & \widetilde{\mathcal{C}_1} \overset{proc}{\sim} [\![\mathcal{C}_0]\!] \\ \vdots & & \downarrow & & \vdots \\ \mathcal{C}_0' & \overset{\mathcal{R}}{\text{---}} & \mathcal{C}_1' & \overset{*}{\underset{\mathbb{T}}{\longrightarrow}} & \widetilde{\mathcal{C}_1'} \overset{proc}{\sim} [\![\mathcal{C}_0']\!] \end{array}$$

If the reduction is $\mathcal{C}_1 \longrightarrow_{\mathbb{T}} \mathcal{C}_1'$, then $\widetilde{\mathcal{C}_1} = \widetilde{\mathcal{C}_1'}$ and $\mathcal{C}_0 = \mathcal{C}_0'$. If it introduces a message $proceed(jp)$, then $\widetilde{\mathcal{C}_1'} \overset{proc}{\sim} \widetilde{\mathcal{C}_1} \overset{proc}{\sim} [\![\mathcal{C}_0]\!]$. If it introduces a message $deploy(pc, ad)$, then $\mathcal{C}_0 \equiv \mathcal{C}_0'$ and $\widetilde{\mathcal{C}_1'} \overset{proc}{\sim} [\![\mathcal{C}_0']\!]$. If it consumes a message $deploy(pc, ad)$, then $\mathcal{C}_0'$ is obtained by applying Rule DEPLOY to the corresponding pointcut/advice pair and $\widetilde{\mathcal{C}_1'} \overset{proc}{\sim} [\![\mathcal{C}_0']\!]$. Otherwise, the reduction consumes a pattern $x.M_\sigma$ and produces a message of the form:

$$w.weave(k, jp).$$

Then, if some aspects match the join point, $\mathcal{C}_0'$ is obtained by applying Rule RED/ASP to $x.M \triangleright P \Vdash^{\varphi} x.M_\sigma$, and if no aspect match, $\mathcal{C}_0'$ is obtained by applying Rule RED/NOASP to $x.M \triangleright P \Vdash^{\varphi} x.M_\sigma$. The fact that the diagram above commutes is a direct consequence of the confluence of $\longrightarrow_{\mathbb{T}}$ and its non-interference with other reductions of the system.

We conclude the proof of the theorem by noting that $[\![\mathcal{C}_0]\!]$ is a normal form for $\longrightarrow_{\mathbb{T}}$, so that $\mathcal{C}_0 \;\mathcal{R}\; [\![\mathcal{C}_0]\!]$. ■

Note that the bisimulation we have defined is not barbed-preserving nor context-closed. This is not surprising as a context would be able to distinguish between the original and translated configuration by using messages sent on auxiliary labels (*weave*, *proceed*, *advice* or *deploy*). But we are interested in equivalent behavior of two closed configurations, not of two terms that can appear in any context, so a simple bisimulation is sufficient.

3.4 Aspect JoCaml

Aspect JoCaml is an implementation of the aspect join calculus on top of JoCaml, an extension of OCaml with join calculus primitives [FLFMS03]. The implementation is directly based on the translation described in Section 3.3. Recall that the decentralized weaving relies on *weaving registries* as a bootstrap mechanism; we discuss some advantages of weaving registries in Section 3.5.2.

While slightly different in the syntax, Aspect JoCaml supports all the functionalities of the aspect join calculus, except for migration, which is not supported in the current implementation of JoCaml (`http://jocaml.inria.fr/`). Using the facilities provided by OCaml, we have also introduced new concepts not formalized in the aspect join calculus, such as classes for both objects and aspects, and the distinction between private and public labels.

This section presents a quick overview of the language through the implementation and deployment of the cache replication example. We then discuss salient points in the implementation.

3.4.1 Overview of Aspect JoCaml

Aspect JoCaml uses directly the class system of OCaml, providing a new `dist_object` keyword to define distributed objects with methods and reactions on public or private labels. For instance, a continuation class that defines a label `k` that expects an integer and prints it to the screen can be defined as:

```
class continuation ip =
  dist_object(self)
  reaction react_k at ip: 'opaque k(n) =
    print_int(n); print_string(" is read\n");0
  public label k
end
```

The label `k` is declared as `public`, meaning that it is visible in a reaction join point. Conversely, a `private` label is not visible, and hence can be neither quantified over nor accessed by aspects. Private labels hence provide another level of encapsulation by hiding patterns, in addition to the possibility to hide reactions discussed in Section 3.2.2. The different per-reaction weaving semantics are specified by a quoted keyword, *e.g.*, `'observable`.

A reaction definition is parametrized by an IP address using `at`. This IP address is meant to be the address of a weaving registry. The parameter `ip` is passed at object creation time, making it possible to choose a different weaving registry for each created continuation object.

The definition of the cache class is given in Figure 3.12 and can be directly inferred from the definition of Section 3.2.2. We omit the code for the dictionary class, which directly uses hash tables provided by the `Hashtbl` OCaml module. A message that creates a dictionary is initially emitted using `spawn` in the `initializer` process.

Aspects are defined as classes with a pointcut and an advice. The instantiation mechanism is identical to that of objects, using the `new` keyword. The cache replication aspect is defined in Figure 3.13. Labels in `Contains` pointcut are handled as strings and boolean pointcut combinators are defined by infix operators `&&&` and `|||`. The only difference with the aspect join calculus in the definition of pointcut/advice is the binding of arguments of the join point. Here, the function `jp_to_arg` must be used with as first argument the label `"l"` of interest and as second argument the join point. This function returns the

```
(* cache class *)
class cache ip dict =
  dist_object(self)
  reaction
     r_get at ip: 'observable
     state(d) & get(k,r) =
              dict#lookup(d,k,r) & state(d)
  or
     r_put at ip: 'observable
     state(d) & put(k,v) =
              dict#update(d,k,v,getDict)
  or
     r_getDict at ip: 'opaque
     getDict(d) = state(d)
  private label state
  public  label get, put, getDict
  initializer spawn dict#create(self#getDict)
end
```

Figure 3.12: Cache class in Aspect JoCaml

```
(* cache replication aspect *)
class replication ip cache =
  dist_object(self)
  reaction
    react_rput at ip: 'opaque
    rput(k,v) = cache#put(k,v)
  public label rput
end

aspect my_asp ip repl =
   pc: Contains x.["put" (k,arg)] &&&
       Not(CausedBy(Contains _.["rput" (_,_)]))
   advice: repl#rput(k,arg) & proceed()
end
```

Figure 3.13: Cache replication aspect in Aspect JoCaml

tuple of arguments passed to `l`, from which each argument can be recovered by explicit pattern matching.

Deployment. Before creating any process, at least one weaving registry must be created and registered to the name server. For instance, the following code creates a permanent weaving registry at IP `12345`:

```
(* create a permanent weaving registry*)
let () =
    let _ = new weaving_registry 12345 in
    while true do Thread.delay 1.0 done
```

Then, a cache replication aspect can be registered to this weaving registry:

```
(* register a cache replication aspect *)
let () =
  let ip   = 12345 in
```

```
  let dict = new Dict.dict ip in
  let buf  = new cache ip dict in
  let repl = new replication ip buf in
  let _    = my_asp ip repl in
  while true do () done
```

Finally, the execution of the cache process defined below is replicated on the machine where the aspect has been deployed:

```
(* a cache process loop *)
let () =
 let ip   = 12345 in
 let dict = new Dict.dict ip in
 let z    = new cache ip dict in
 let k    = new continuation ip in
 for arg = 1 to 10 do
   spawn z#put("key",arg);
   spawn z#get("key",k#k)
 done;
```

3.4.2 Implementation

We now briefly discuss some elements of the Aspect JoCaml implementation.

Architecture. An Aspect JoCaml file is translated into a JoCaml file and then compiled using the JoCaml compiler. To simplify the parser, there are { ... } separators for plain JoCaml code (for clarity, those separators have been omitted in Figure 3.13). While these separators clutter the code, they have the advantage that new features of JoCaml or OCaml can be directly back ported to Aspect JoCaml.

A more advanced solution would be to use Camlp5, the preprocessor-pretty-printer of OCaml, to produce a type-safe translation. Unfortunately, compatibility issues between Camlp5 and JoCaml forbids this solution at the moment.

Typing issues. As the code produced is compiled using JoCaml, everything needs to be typed. Sometimes, this requires type annotations in class definitions when dealing with parametric polymorphism.

However, as mentioned in the JoCaml manual: "communications through the name server are untyped. This weakness involves a good programming discipline."[MM12] On the one hand, this limitation of distributed programming in OCaml simplifies the task of creating a list of aspects of different types. On the other hand, to avoid type errors at runtime, an anti-unification mechanism has to be developed to guarantee type safe application of aspects [TFT13].

Static/dynamic pointcuts. The weaving registry is responsible for bootstrapping the communication between weavers and aspects. This is performed by adding aspects to the list of current aspects connected to the weaver. But part of communications between weavers and aspects can be avoided. Indeed, it is sometimes possible to statically decide whether a pointcut can match a join point coming from a given weaver. If the pointcut can never match, the weaving registry does not need to register the aspect to the weaver.

To that end, our implementation differentiates between the static and dynamic parts of a join point. The static part is used at registration time, whereas the dynamic part is used during runtime weaving.

Bounded depth of causality tree. An optimized, scalable management of the causality tree discussed in Section 3.2.3 is a challenging research challenge. The current implementation is naive, keeping track of every causal match. This means that the causality tree may grow unboundedly. Therefore, a bounded version of the causality tree ought to be implemented. This raises the issue of deciding the size of the tree, because dealing with a bounded tree changes the semantics of pointcut matching. It remains to be studied how existing optimization techniques for control flow pointcuts [MKD03] and trace-based matching [ACH+06] could be adapted to the general setting of the causality tree.

3.5 Discussion

3.5.1 Synchronous aspects

A particularity of aspects compared to traditional event handling is the possibility to advise around join points and therefore have the power to proceed the original computation, either once, several times, or not at all. Doing so requires careful thinking about the synchronization of advices. The semantics we have presented corresponds to *asynchronous* reactions, in which all advices that match are triggered asynchronously. We could devise a weaving semantics that rather reflects the one of AspectJ by chaining implicitly advices and invoking them in a *synchronous* manner. First, this presents the issue of choosing the order in which advices are chain, which is not clear in an asynchronous setting. Second, the synchronous semantics can be encoded by an explicit chaining of advices and thus is not a primitive operations. For those two reasons, we have decided not to integrate a synchronous reaction in the semantics.

Also, note that the semantics of aspect weaving relies on the currently-deployed aspects. As we have seen, deployment is asynchronous, which means that to be sure that an aspect is in operation at a given point a time, explicit synchronization has to be setup. This design is in line with the asynchronous chemical semantics of the join calculus. For instance, the same non-determinism occurs in the definition of an object, in which the initialization process is not guaranteed to be completed before the object process starts executing. In case such sequentiality is needed, it has to be manually encoded.

3.5.2 Distributed aspect deployment

Distributed aspect deployment is a complex task, for which several different policies can be conceived. This is reflected in the different designs and choices of specific distributed AOP systems, such as DJcutter (one central aspect server) [NCT04], AWED (aspects are either deployed on all hosts, or only on their local host of definition) [BNSV+06], and ReflexD (distributed aspect repositories to which base programs are connected at start-up time) [TT06], among others.

The weaving registries we propose are a simple and flexible abstraction that captures the need for precisely specifying the distributed deployment of aspects. This mechanism is very flexible topologically (*e.g.*, the case where all aspects see all computation correspond to one global weaving registry to which all weavers and aspects are registered), and allows for fine-grained policies that go beyond existing work. For instance, some weaving registries (called *closed*) may be initiated with a fixed number of aspects deployed, and subsequently reject any new aspect registration requests. Other registries (called *open*) may accept dynamic aspect registration requests, with the restriction that these aspects will not be able to react to the computation of previously-registered rules. A weaving

registry policy may further specify that only weavers of a specific kind are accepted, such as observable reactions (Section 3.2.2).

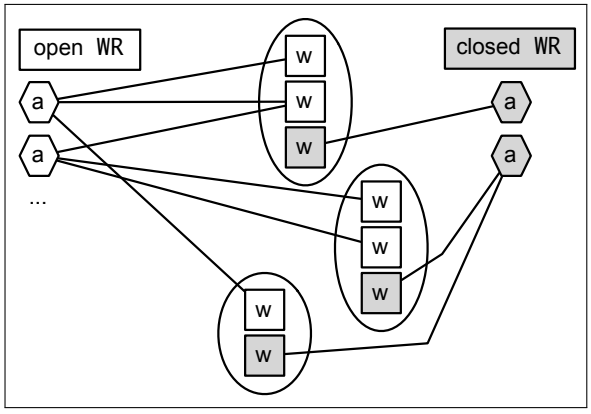


Figure 3.14: Objects registering their public weavers to an open weaving registry (white), and the weavers of their sensitive reactions to a closed registry with only two trusted aspects (grey).

Figure 3.14 illustrates some of the topological flexibility offered by weaving registries and their policies: objects can register their reaction weavers in different registries with specific policies. For instance, suppose a closed weaving registry reg_c with only a cache replication aspect and an open weaving registry reg_o. The following definition of the cache example guarantees that only cache replication can have access to the cache history (we extend the syntax of reactions with an exponent $M \triangleright^{reg} P$ to express that the reaction is registered in the weaving registry reg):

$$
\begin{array}{l}
\mathsf{obj}\ c = \ put(k,v)\ \&\ state(d) \circledcirc^{reg_c} dict.update(d,k,v,c) \\
\quad\ \mathsf{or}\ \ get(k,r)\ \&\ state(d) \circledcirc^{reg_o} dict.lookup(d,k,r)\ \&\ c.state(d) \\
\quad\ \mathsf{or}\ \ getDict(d) \blacktriangleright^{reg_o} c.state(d) \\
\mathsf{in}\ \ dict.create(c)
\end{array}
$$

The design space of distributed deployment policies is wide and its exhaustive exploration is left open.

3.6 Related work

We first discuss work related to the formal semantics of aspects, and then relate to existing distributed aspect languages and systems.

3.6.1 Formal semantics of aspects

There is an extensive body of work on the semantics of aspect-oriented programming languages (*e.g.*, [WKD04, CL06, DTK06, JJR06, DFES10]). These languages adopt either the lambda calculus or some minimal imperative calculus at their core. To the best of our knowledge, this work is the first to propose a chemical semantics of aspects. In addition, none of the formal accounts of AOP considers distributed aspects. Among practical distributed aspect systems, only AWED exposes a formal syntax; the semantics of the language is however only described informally [BNSV+06].

The approach of starting from a direct semantics with aspects, and then defining a translation to a core without aspects and proving the correctness of the transformation is also used by Jagadeesan et al., in the context of an AspectJ-like language [JJR06].

3.6.2 Distributed aspect languages and systems

We now compare specific features of practical distributed aspect languages and systems—in particular JAC [PSD+04], DJcutter [NCT04], ReflexD [TT06], and AWED [BNSV+06]—and relate them to the aspect join calculus.

Quantification. Remote pointcuts were first introduced in DJcutter and JAC, making it possible to specify on which hosts joint points should be detected. Remote pointcuts are also supported in AWED, ReflexD, and in the aspect join calculus, in a very similar fashion. Remote pointcuts can be seen as a necessary feature for distributed AOP (as opposed to using standard AOP in a distributed setting).

Hosts. Remote pointcuts bring about the necessity to refer to execution hosts. In DJcutter and AWED, hosts are represented as strings, while in ReflexD they are reified as objects that give access to the system properties of the hosts. The host model in ReflexD is therefore general and expressive, since host properties constitute an extensible set of metadata that can be used in the pointcuts to denote hosts of interest. In the aspect join calculus, we have not developed locations beyond the fact that they are first class values. A peculiarity is that locations are organized hierarchically, and can possibly represent finer-grained entities than in existing systems (for instance, a locality can represent an actor within a virtual machine within a machine). A practical implementation should consider the advantages of a rich host metadata model as in ReflexD. AWED and ReflexD support dynamically-defined groups of hosts, as a means to deal with the distributed architecture in a more abstract manner than at the host level.

Weaving semantics. Most distributed AOP languages and systems adopt a synchronous aspect weaving semantics. This is most probably due to the fact that the implementation is done over Java/RMI, in which synchronous remote calls is the standard. Notably, AWED supports the ability to specify that some advices should be run asynchronously. The aspect join calculus is the dual: the default is asynchronous communication, but we can also express synchronous weaving (Section 3.2.2). In addition, we have developed the ability to customize the weaving semantics on a per-reaction basis. An interesting consequence of this granularity is that we are able to express opaque and observable reactions. Both kinds of reactions support stronger encapsulation and guarantees in presence of aspects, and therefore fit in the line of work on modular reasoning about aspects [BTI14, OSC10, SPAK10, TFT13].

Advanced quantification. DJcutter, AWED, and ReflexD support reasoning about distributed control flow, in order to be able to discriminate when a join point in the (distributed) flow of a given method call. AWED also support state-machine-like pointcuts, called stateful aspects, which are able to match sequences of events that are possibly unrelated in terms of control flow. However, stateful aspects per se do not make it possible to reason about causality; additional support is needed, for instance as developed in WeCa [LTD12]. In Section 3.2.3, we describe how join points can capture their causality links, which can then be used for pointcut matching. While the synchronous communication pattern can be recognized in order to support a similar notion of distributed

control flow, the causality tree model is much more general. An interesting venue for future work is to develop a temporal logic for pointcuts that can be used to reason precisely about causality. Temporal logic has been used in some aspect-oriented systems to perform semantic interface conformance checks [BS06]. Causality in widely-asynchronous (distributed) contexts is a topic of major interest. It would be interesting to study how our approach relates to the notion of causality in the π-calculus proposed by Curti et al. in the context of modeling biochemical systems [CDB03].

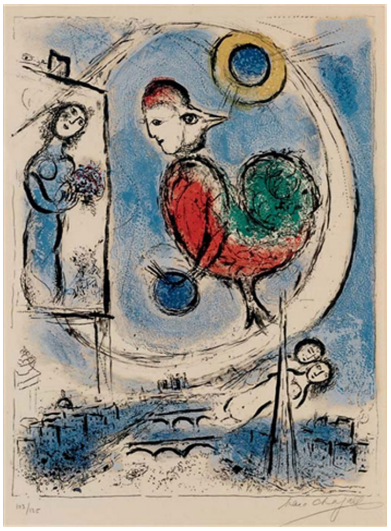
Aspect deployment. DJcutter adopts a centralized architecture with an aspect host where all aspects reside and advices are executed. JAC allows distributed aspect deployment to various containers with a consistency protocol between hosts, to ensure a global view of the aspect environment. Both AWED and ReflexD adopt a decentralized architecture, in which it is possible to execute advices in different hosts: multiple parallel advice execution in specific hosts is possible, and programmers can control where aspects are deployed. ReflexD is more flexible than AWED in the localization of advices and in deployment, by supporting stand-alone aspect repositories to which a Reflex host can connect. The weaving registries mechanism we have described in Section 3.5.2 subsumes these mechanisms, and also adds support for controlling the openness of the distributed architecture.

JAC, AWED and Reflex support dynamic undeployment of aspects. While we have not introduced undeployment in this chapter, it is trivial to add it to the core calculus. More interesting, in previous work we explore structured deployment through *scoping strategies* [TFD+10]. Scoping strategies make it possible to specify the computation that is exposed to a given aspect in a very precise manner. The model of scoping strategies relies on per-value and per-control-flow propagation of aspects; it would be not trivial, but interesting, to study how these strategies can be adapted to a chemical setting.

Parameter passing. In Java, remote parameter passing is by-copy, unless for remote objects that are passed by-reference. ReflexD allows to customize the remote parameter passing strategy for each parameter passed to a remote advice. The join calculus has a by-reference strategy, where names act as references. It would be possible to add a by-copy mechanism in the aspect join calculus, by adding a rule to clone named definitions.

Chapter 4

Partial Type Equivalences for Verified Dependent Interoperability*



Le Coq sur Paris, Chagall (1958)

Contents

*This is joint work with Pierre-Évariste Dagand and Éric Tanter [DTT16].

Dependent interoperability is a pragmatic approach to building reliable software systems, where the adoption of dependent types may be incremental or limited to certain components. The sound interaction between both type disciplines relies on a *marshalling* mechanism to convert values from one world to the other, as well as *dynamic checks*, to ensure that the properties stated by the dependent type system are respected by the simply-typed values injected in dependent types.[1]

Following [OSZ12], we illustrate the typical use cases of dependent interoperability using the simple example of simply-typed lists and dependently-typed vectors. For conciseness, we fix the element type of lists and vectors and use the type synonyms $\texttt{List}_{\mathbb{N}}$ and $\texttt{Vec}_{\mathbb{N}}$ `n`, where `n` denotes the length of the vector.

Using a simply-typed library in a dependently-typed context. One may want to reuse an existing simply-typed library in a dependently-typed context. For instance, the list library may provide a function $\texttt{max} : \texttt{List}_{\mathbb{N}} \to \mathbb{N}$ that returns the maximum element in a list. To reuse this existing function on vectors requires lifting the `max` function to the type $\forall\ \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{n} \to \mathbb{N}$. Note that this scenario only requires losing information about the vector used as argument, so no dynamic check is needed, only a marshalling to reconstruct the corresponding list value. If the simply-typed function returns a list, *e.g.*, $\texttt{rev} : \texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$, then the target dependent type might entail a dynamic check on the returned value.

Using a dependently-typed library in a simply-typed context. Dually, one may want to apply a function that operates on vectors to plain lists. For instance a sorting function of type $\forall\ \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{n} \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$ could be reused at type $\texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$. Note that this case requires synthesizing the index `n`. Also, because the simply-typed argument flows to the dependently-typed world, a dynamic check might be needed. Indeed, the function $\texttt{tail} : \forall\ \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{(n+1)} \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$, should trigger an error if it is called on an empty list. On the return value, however, no error can be raised.

Verifying simply-typed components. One can additionally use dependent interoperability to dynamically verify properties of simply-typed components by giving them a dependently-typed interface and then going back to their simply-typed interface, thereby combining both scenarios above. For instance, we can specify that a function $\texttt{tail} : \texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$ should behave as a function of type $\forall\ \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{(n+1)} \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$ by first lifting it to this rich type, and then recasting it back to a simply-typed function `tail'` of type $\texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$. While both `tail` and `tail'` have the same type and "internal" definition, `tail'` will raise an error if called with an empty list; additionally, if the argument list is not empty, `tail'` will dynamically check that it returns a list that is one element smaller than its input. This is similar to dependent contracts in untyped languages [FF02].

[1] In this chapter, we use the term "simply typed" to mean "non-dependently typed", *i.e.*, we do not rule out parametric polymorphism.

Program extraction. Several dependently-typed programming languages use program extraction as a means to obtain (fast(er)) executables. Coq is the most prominent example, but more recent languages like Agda, Idris, and F* also integrate extraction mechanisms, at different degrees (*e.g.*, extraction in F* is the only mechanism to actually run programs, while in Agda it is mostly experimental at this point).

Dependent interoperability is crucial for extraction, if extracted components are meant to openly interact with other components written in the target language. While [TT15] address the question of protecting the extracted components from inputs that violate conditions expressed as subset types in Coq[2], the situation can be even worse with type dependencies, because extracting dependent structures typically introduces unsafe operations; hence invalid inputs can easily produce segmentation faults.

Consider the following example adapted from the book *Certified Programming with Dependent Types* [Chl13], in which the types of the instructions for a stack machine are explicit about their effect on the size of the stack:

```
Inductive dinstr: ℕ → ℕ → Set :=
| IConst: ∀ n, ℕ → dinstr n (S n)
| IPlus: ∀ n, dinstr (S (S n)) (S n).
```

An `IConst` instruction operates on any stack of size `n`, and produces a stack of size `(S n)`, where `S` is the successor constructor of ℕ. Similarly, an `IPlus` instruction consumes two values from the stack (hence the stack size must have the form `(S (S n))`, and pushes back one value. A dependently-typed stack of depth `n` is represented by nested pairs:

```
Fixpoint dstack (n: ℕ): Set :=
 match n with
  | O ⇒ unit
  | S n' ⇒ ℕ × dstack n'
 end.
```

The `exec` function, which executes an instruction on a given stack and returns the new stack can be defined as follows:

```
Definition exec n m (i: dinstr n m):
  dstack n → dstack m :=
 match i with
  |IConst n ⇒ fun s ⇒ (n, s)
  |IPlus ⇒ fun s ⇒
   let (arg1, (arg2, s')) := s in (arg1 + arg2, s')
 end.
```

Of special interest is the fact that in the `IPlus` case, the stack `s` is deconstructed by directly grabbing the top two elements through pattern matching, without having to check that the stack has at least two elements— this is guaranteed by the type dependencies.

Because such type dependencies are absent in OCaml, the `exec` function is extracted into a function that ignores its stack size arguments, and relies on unsafe coercions:

```
(* exec: int → int → dinstr → dstack → dstack *)
let exec _ _ i s =
 match i with
 | IConst (n, _) → Obj.magic (n, s)
 | IPlus _ →
```

[2] In Coq terminology, a subset type is a type refined by a proposition—this is also known in the literature as refinement type [RKJ08].

```
  let (arg1, s1) = Obj.magic s in
  let (arg2, s2) = s1 in Obj.magic ((add arg1 arg2), s2)
```

The `dstack` indexed type from Coq cannot be expressed in OCaml, so the extracted code defines the (plain) type `dstack` as:

```
type dstack = Obj.t
```

where `Obj.t` is the abstract internal representation type of any value. Therefore, the type system has in fact no information at all about stacks: the unsafe coercion `Obj.magic` (of type $\forall a \forall b. a -> b$) is used to convert from and to this internal representation type. The dangerous coercion is the one in the `IPlus` case, when coercing `s` to a nested pair of depth at least 2. Consequently, applying `exec` with an improper stack yields a segmentation fault:

```
# exec 0 0 (IPlus 0) [1;2];;
- : int list = [3]
# exec 0 0 (IPlus 0) [];;
Segmentation fault: 11
```

Dependent interoperability helps in such scenarios by making it possible to lift dependent structures—and functions that operate on them—to types that are faithfully expressible in the type system of the target language in a sound way, *i.e.*, embedding dynamic checks that protects extracted code from executing unsafe operations under violated assumptions.[3] We come back to this stack machine example and how to protect the extracted `exec` function in Section 4.4.

Contributions In this work, we present a verified dependent interoperability layer for Coq that exploits the notion of type equivalence from Homotopy Type Theory (HoTT). In particular, our contributions are the following:

- Using type equivalences as a guiding principle, we give a unified treatment of *(partial) type equivalences* between programs (Section 4.1). Doing so, we build a conceptual as well as practical framework for relating indexed and simple types;
- By carefully segregating the computational and logical content of indexed types, we introduce a notion of *canonical equivalence* (Section 4.2) that identifies first-order transformations from indexed to simple datatypes. In particular, we show that an indexed type can be seen as the combination of its underlying computational representation and a runtime check that its associated logical invariant holds;
- To deal with programs, we extend the presentation to a higher-order setting (Section 4.3). Using the type class mechanism of Coq, we provide a generic library for establishing partial type equivalences of dependently-typed programs;
- Finally, we illustrate our methodology through a concrete application: extracting an interoperable, certified interpreter (Section 4.4). Aside from exercising our library, this example is also a performance in homotopic theorem proving.

This chapter is thus deeply entrenched at the crossroad between mathematics and programming. From the former, we borrow and introduce some homotopic definitions as well as proof patterns. For the latter, we are led to design interoperable—yet safe—programs and are willing to trade static safety against runtime checks.

[3] Note that some unsafe executions can be produced by using impure functions as arguments to functions extracted from Coq—because referential transparency is broken. Designing an adequate protection mechanism to address such scenarios is a separate, interesting research challenge.

4.1 Partial Type Equivalences

Intuitively, dependent interoperability is about exploiting a kind of equivalence between simple and indexed types. This section formally captures such an equivalence relation, which we call *partial* because, as illustrated previously, some runtime errors might occur when crossing boundaries.

We use Coq as both a formalization vehicle and an implementation platform. We make extensive use of *type classes* [WB89] in order to define abstract structures and their properties, as well as relations among types. For instance, a partial type equivalence is a type class, whose instances must be declared in order to state an equivalence between specific types, such as $\texttt{Vec}_{\mathbb{N}}$ `n` and $\texttt{List}_{\mathbb{N}}$. As opposed to Haskell, type classes in Coq [SO08] can express arbitrary properties that need to be proven when declaring instances—for instance, the monad type class in Coq *defines and imposes* the monad laws on each instance.

In this section we progressively define the notion of partial type equivalences in a general manner. We apply partial type equivalences to the dependent interoperability context in Section 4.2, focusing on first-order equivalences between types of data structures. Later, in Section 4.3, we build higher-order partial type equivalences to interoperate between functions that manipulate such structures.

4.1.1 Type Equivalence

The notion of type equivalence offers a conceptual framework in which to reason about the relationships between types. Following intuitions coming from homotopy theory, a type equivalence between two types `A` and `B` is defined by a function `f: A → B` such that there exists a function `e_inv : B → A`, with proofs that it is both its left and right inverse together with a compatibility condition between these two proofs [Uni13]. This definition plays a central role in Homotopy Type Theory (HoTT), as it is at the heart of the univalence axiom.

In this chapter, we exploit type equivalence as a means to (constructively) state that two types "are the same". In Coq, this amounts to the following type class definition:[4]

```
Class IsEquiv (A B : Type) (f:A → B) := {
  e_inv : B → A ;
  e_sect : e_inv ∘ f == id;
  e_retr : f ∘ e_inv == id;
  e_adj  : ∀ x, e_retr (f x) = ap f (e_sect x)
}.
```

The properties `e_sect` and `e_retr` capture the fact that `e_inv` is the inverse of `f`. The definitions use the identity function `id`, and point-wise equality between functions `==`. The extra coherence condition `e_adj` ensures that the equivalence is uniquely determined by the function `f`, that is, being an equivalence is proof-irrelevant (where `ap f` is the functorial action of `f`, transporting an equality between `x` and `y` to an equality between `f x` and `f y`).

4.1.2 Towards Partial Type Equivalence: The Cast Monad

As illustrated in the introduction, lifting values from simple to indexed types can fail at runtime. Thus, the type equivalences we are interested in are *partial*. To denote—and

[4] Adapted from: `http://hott.github.io/HoTT/coqdoc-html/HoTT.Overture.html#IsEquiv`

reason about—partial functions, we resolve to use pointed sets [Hyl91]. In Coq, those are naturally modeled by working in the Kleisli category of the `option` monad, with a `None` constructor to indicate failure, and a `Some` constructor to indicate success.

We thus define a specific `Cast` monad, which is essentially the `option` monad.[5] We use the harpoon notation ⇀ to denote a (partial) function in the `Cast` monad: `Notation " A ⇀ B" := (A → Cast B)`. The `Cast` monad is characterized by its identity `creturn` and binder `cbind`. We use the traditional do-notation. For instance, function composition in the corresponding Kleisli category, denoted $\circ_K$, is defined as follows:[6]

```
Definition kleisliComp {A B C : Type}:
 (A ⇀ B) → (B ⇀ C) → (A ⇀ C) :=
 fun f g a ⇒ b ← f a ; g b.
Notation "g ∘_K f" := (kleisliComp f g).
```

We thus closely model the denotational objects we are interested in (here, partial functions). Crucially, the nature of these objects is reflected at the type-level: types play a guiding role in Section 4.1.3 below, where we lift the notion of type equivalence to the partial setting.

4.1.3 Partial Type Equivalence

In this section, we aim at reconciling the general notion of type equivalence with the potential for errors, as modeled by the `Cast` monad. To do so, we observe that the `Cast` monad induces a preorder. This naturally leads us to generalize the equivalence relation to operate on preorders.

Cast as a preorder with a least element. The notion of preorder with a least element is naturally defined in Coq with the following type class:

```
Class PreOrder⊥ (A:Type) :=
 { rel : A → A → Prop where "x ⪯ y" := (rel x y);
  ⊥ : A;
  rel_refl : ∀ x, x ⪯ x ;
  rel_trans : ∀ x y z, x ⪯ y → y ⪯ z → x ⪯ z;
  ⊥_is_least : ∀ a, ⊥ ⪯ a
 }.
```

The `Cast` monad induces a preorder which corresponds to equality on success values, and considers `None` as the least element of the ordering. More precisely:

```
Instance PreOrderCast A : Preorder⊥ (Cast A) :=
 {| rel := fun a a' ⇒ match a with
               | Some _ ⇒ a = a'
               | None  ⇒ True
               end;
  ⊥ := None |}.
```

Any preorder on the codomain of two functions gives us a way to compare these functions pointwise.[7]

[5] We discuss some specificities of the `Cast` monad in Section 4.4.5.

[6] In Coq, parameters within curly braces are implicitly resolved.

[7] In Coq, back-quoted parameters are nameless. Records differ from type classes in that they are not involved in (implicit) instance resolution; other than that, type classes are essentially records [SO08].

```
Instance Preorder⊥_fun (A: Type) (B: A → Type)
 `{∀ a, Preorder⊥ (B a)} : Preorder⊥ (∀ a, B a) :=
 {| rel := fun f g ⇒ ∀ a, f a ≼ g a;
    ⊥ := fun a ⇒ ⊥ |}.
```

Monotone functions. We must now generalize type equivalence from types equipped with an equality to types equipped with a preorder. To witness such a partial type equivalence, we shall ask for a *monotonic* function, *i.e.*, a function that preserves the preorder relation (and the least element).

```
Record monotone_function X Y `{Preorder⊥ X}
 `{Preorder⊥ Y} := Build_Mon
   { f_ord :> X → Y ;
     mon :∀ x y, x ≼ y → f_ord x ≼ f_ord y;
     mon_p : f_ord ⊥ ≼ ⊥
   }.
Notation "X ⟶ Y" := (monotone_function X Y).
```

Monotonicity is expressed through the functorial action `f.(mon)`, thus following and generalizing the functorial action `ap f` of (total) type equivalences (Section 4.1.1). We use a type class definition `Functor` to overload the notation `ap` of functorial action. The `:>` notation in the field `f_ord` declares an implicit coercion from `X ⟶ Y` to `X → Y`: we can transparently manipulate monotone functions as standard functions.

Partial equivalence. We now capture the notion of a partial equivalence between two preorders `A` and `B`. The definition of `IsPartialEquiv` is directly derived from its total counterpart, by replacing functions with monotone functions, and equality by the preorder relation `≼`.

```
Class IsPartialEquiv (A B : Type) (f:A ⟶ B)
`{Preorder⊥ A} `{Preorder⊥ B} := {
 pe_inv : B ⟶ A ;
 pe_sect : pe_inv ∘ f ≼ id;
 pe_retr : f  ∘ pe_inv ≼ id;
 pe_adj   : ∀ x, pe_retr (f x) = ap f (pe_sect x)
}.
```

4.1.4 Partial Type Equivalence in the Kleisli Category

The preorder over `Cast` types yields the expected notion of type equivalence in the Kleisli (bi-)category defined in Section 4.1.2. Composition amounts to monadic composition $\circ_K$, and identity to monadic identity `creturn`. We substantiate this intuition by specifying a monadic equivalence to be:

```
Class IsPartialEquivK (A B : Type) (f:A ⇀ B) := {
 pek_inv : B ⇀ A ;
 pek_sect : pek_inv ∘K f ≼ creturn ;
 pek_retr : f ∘K pek_inv ≼ creturn;
 pek_adj : ∀ x,
    ((pek_sect ∘V (id2 f)) ∘H idL f) x =
    (α f pek_inv f ∘H ((id2 f) ∘V pek_retr) ∘H idR f) x
}.
```

Note that the definition of `pek_adj` is more complicated than for partial equivalences, as explained in Section 4.1.5 below.

As a sanity check, we can prove that lifting an equivalence yields a partial type equivalence in the Kleisli category, meaning in particular that `pek_adj` is a conservative extension of `e_adj` to the Kleisli category.

```
Definition EquivToPartialEquivK A B (f :A → B) :
 IsEquiv f → IsPartialEquivK (clift f).
```

4.1.5 A (Bi-)Categorical Detour

The various notions of equivalence presented above (total, partial, and partial in the Kleisli category of the `Cast` monad) follow a common pattern—they are all instances of the concept of adjunction in a bicategory, coming from the seminal work of [Bén67]. Recall that 2-categories generalize categories by introducing a notion of 2-cells, *i.e.*, morphisms between morphisms, but letting compatibility laws hold strictly. In the setting of type theory, the fact that compatibility laws are strict means that they hold definitionally by conversion in the system. Bicategories generalize 2-categories by allowing compatibility laws not to be defined strictly, but up-to an invertible 2-cell.

Relaxing compatibility laws up-to invertible 2-cells is not necessary to describe (total) type equivalence because associativity and identity laws hold strictly on functions between types, as they are directly captured by β-reduction.

For partial type equivalence, strict laws for composition also hold because we are dealing with proofs of monotonicity that are irrelevant in the sense that they are stated on a notion of preorder that lives in `Prop`. Note that we could have defined a proof-relevant preorder and monotonicity condition, in which case the need to go to a bicategorical setting would have manifested itself at this stage.

When considering the Kleisli category induced by the `Cast` monad, however, it is not possible to avoid bicategories anymore because, for instance, associativity of Kleisli composition does not hold strictly. Indeed, the different order in which arguments are evaluated (*i.e.*, in which effects are performed) matters, and so associativity only holds up to a proof term. That is, there is a term to make explicit the associativity of composition (we express it from left-to-right but the converse also holds):

```
Definition α {X Y Z T: Type} (f:X ⇀ Y) (g:Y ⇀ Z) (h:Z ⇀ T):
  h ∘K (g ∘K f)) ⪯ (h ∘K g) ∘K f.
```

In the same way, we need to exhibit the usual morphisms:

```
idR f : creturn ∘K f ⪯ f                          (right identity law)
idL f : f ∘K creturn ⪯ f                          (left identity law)
∘V : f ⪯ f' → g ⪯ g'
        → g ∘K f ⪯ g' ∘K f'                       (vertical composition)
∘H : e ⪯ f → f ⪯ g → e ⪯ g                        (horizontal composition)
id2 f : f ⪯ f                                     (2-identities)
```

It follows that the definition of `pek_adj` is merely the expected formulation of the compatibility of an adjunction in a bicategory.

4.2 Partial Type Equivalence for Dependent Interoperability

We now exploit partial type equivalences to setup a verified framework for dependent interoperability. In this context, we are specifically interested in partial equivalences between indexed types, such as $\texttt{Vec}_{\mathbb{N}}$, and simple types, such as $\texttt{List}_{\mathbb{N}}$. We call this kind of partial type equivalence a *dependent equivalence*.

A major insight of this work is that a dependent equivalence can be defined by composition of two different kinds of type equivalences, using subset types as intermediaries:

- a *total* equivalence between indexed types and subset types of the form `{c:C & P c}` whose logical content `P`—*i.e.*, static invariants—is carefully quarantined from their computational content `C`—*i.e.*, runtime representation;
- a *partial* equivalence between subset types and simple types, in the Kleisli category of the `Cast` monad.

The resulting dependent equivalence is therefore also a partial type equivalence in the Kleisli category.

For instance, to establish the equivalence between $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$, we exploit the subset type $\texttt{\{l:List}_{\mathbb{N}}\texttt{ \& length l = n\}}$, which captures the *meaning* of the index of $\texttt{Vec}_{\mathbb{N}}$.

4.2.1 Partial Equivalence Relations

The equivalence classes defined in Section 4.1 characterize a *specific function* as witnessing an equivalence between two types, thereby allowing different functions to be thus qualified.

Following the Coq HoTT library, we define the `Equiv` record type to specify that *there exists* an equivalence between two types `A` and `B`, denoted `A ≃ B`. The record thus encapsulates the equivalence function `e_fun` and defines a type relation:

```
Record Equiv (A B : Type) := {
  e_fun : A → B ;
  e_isequiv : IsEquiv e_fun
}.
Notation "A ≃ B" := (Equiv A B).
```

For partial type equivalences in the Kleisli category of the `Cast` monad, we similarly define the record type and notation:

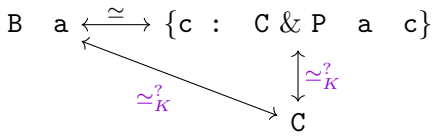
```
Record PartialEquivK (A B : Type) := {
  pek_fun : A ⇝ B;
  pek_isequiv: IsPartialEquivK pek_fun
}
Notation "A ≃?K B" := (PartialEquivK A B).
```

For dependent interoperability, however, we want to consider *canonical* dependent equivalences between index and simple types, so as to automate the lifting between both worlds. This is particularly important to perform complex, automatic lifting of functions, as described in Section 4.3. For instance, lifting a function that operate on $\texttt{Vec}_{\mathbb{N}}$ to an equivalent function that operates on $\texttt{List}_{\mathbb{N}}$ implies "looking up" the canonical equivalence between these types.

Technically, this means that canonical partial equivalences need to be presented to Coq as a type class, in order to take advantage of the automatic instance resolution mechanism they offer:

```
Class CanonicalPartialEquiv (A B : Type) := {
 pe_fun : A ⟶ B;
 pe_isequiv: IsPartialEquiv pe_fun
}
Notation "A ≃? B" := (CanonicalPartialEquiv A B).
```

The strategy to establish dependent equivalences via subset types can be depicted as follows, for a given index a:



The following subsections describe the two (orthogonal) equivalences below, before defining their (diagonal) composition.

In the diagram above, and the rest of this chapter, we adopt the following convention: the type index is `A : Type`, the type family is `B : A → Type`, the plain type is `C : Type`, and the logical proposition capturing the meaning of the index is `P : A → C → Type`.

Remark: HProp/HSet vs. Prop/Set The reader might expect `P` to end in `Prop`, not `Type`. One of the initial goal of the sort `Prop` of Coq was to capture proof irrelevant properties that can be erased during extraction. This *syntactic* characterization is however not correct. On the one hand, it is too restrictive because some properties are proof irrelevant but cannot be typed in `Prop` [MS08]. On the other hand, there are elements of `Prop` that cannot be proven to be proof irrelevant. The most famous example of such an element is the equality type. Indeed, for every type `A:Type` and elements `a,b:A`, we have `a=b : Prop` in Coq, but proving that `a=b` is irrelevant is independent from the theory of Coq as it corresponds to the Uniqueness of Identity Proofs (UIP) axiom.

Therefore, we face two possible design choices. We could consider propositions in `Prop` and datatypes in `Set`, assuming UIP—which could be seen as controversial. Instead of relying on an axiom, we choose to require proof irrelevance *semantically* whenever it is needed. This semantic characterization of types with a proof-irrelevant equality is specified by the type class `IsHProp` as introduced in the Coq HoTT library:

```
Class IsHProp (T: Type) := {is_hprop : ∀ x y:T, x = y}.
```

In the same way, types that semantically form sets can be characterized by the type class `IsHSet`:

```
Class IsHSet X := {isHSet :> ∀ (a b : X), IsHProp (a = b)}.
```

In the rest of the code, we (abusively) write `T : HProp` for a type `T : Type` for which there exists an instance of `IsHProp T` (and similarly for `HSet`). Therefore, the logical proposition capturing the meaning of the index is hereafter written `P : A → C → HProp`. Similarly, since simple types represent data structures, which are sets, we write `C : HSet`.

4.2.2 Equivalence Between Indexed and Subset Types

The first step is a total equivalence between indexed types and subset types. In our dependent interoperability framework, this is the only equivalence that the programmer has to manually establish and prove. For instance, to relate lists and vectors, the programmer must establish that, for a given `n`:

```
Vec_ℕ n ≃ { l : List_ℕ & length l = n }
```

Recall from Section 4.1.1 that establishing this total equivalence requires providing two functions:

```
vector_to_list n: Vec_ℕ n → { l : List_ℕ & length l = n }
list_to_vector n: { l : List_ℕ & length l = n } → Vec_ℕ n
```

These functions capture the computational content of the conversion between the two data structures.[8] The programmer must also prove that they are inverse of each other. In addition, she needs to prove the `e_adj` property. This coherence property is generally quite involved to prove. We come back to the question of proving the coherence in Section 4.2.5, and we discuss some useful proof techniques for type conversions in Section 4.4.6.

4.2.3 Equivalence Between Subset and Simple Types

The second equivalence we exploit is a *partial* type equivalence in the Kleisli category of the `Cast` monad between subset types and simple types such as, for a given `n`:

```
{ l : List_ℕ & length l = n } ≃?_K List_ℕ
```

Obviously, going from the subset type to the simple type never fails, as it is just the first projection of the underlying dependent pair π_1. However, the other direction is not always possible: it depends if the given `n` is equal to the length of the considered list.

Recently, [TT15] developed an approach in Coq to cast a value of any simple type `C` to a value of a subset type `{c:C & P c}` for any decidable predicate `P`. Using decidability, the authors perform the type cast through a runtime check, relying on an axiom to capture the potential for cast failure. In the monadic setting we adopt here, there is no need for axioms anymore, and their technique amounts to establishing a partial type equivalence `{c:C & P c}` $\simeq^?_K$ `C`. We capture this partial type equivalence between subset types and simple types with the following instance:

```
Definition Checkable_PEquivK (C : HSet) (P : C → HProp)
`{∀ c, Checkable (P c)} : {c:C & P c} ≃?_K C :=
 {| pek_fun := (clift π1 : {c:C & P c} ⇀ C);
    pek_isequiv := {| pek_inv := to_subset |}|}.
```

Instead of imposing actual *decidability*, the `Checkable` type class (defined in Appendix) only asks for the predicate to be *checkable*, *i.e.*, there must exist a decidable, sound approximation. We also demand proof irrelevance of `P` (via `HProp`).

The equivalence function `pek_fun` is the (lifting of the) first projection function π_1 (the type ascription is necessary to help the Coq type inference algorithm). The inverse function is the `to_subset` function below, which is essentially a monadic adaptation of the cast operator of [TT15]:

```
Definition to_subset {C : HSet} {P : C → HProp}
 `{∀ c, Checkable (P c)}: C ⇀ ({c:C & P c}) :=
   fun c ⇒
    match dec checkP with
    | inl p ⇒ Some (c; convert p)
    | inr _ ⇒ None
    end.
```

[8] Note that a programmer may very well choose to define a conversion that reverses the elements of the structure. As long as the equivalence is formally proven, this is permitted by the framework; any lifting that requires the equivalence uses the user-defined canonical instance.

`to_subset` applies the sound approximation decision procedure for the embedded logical proposition. If it succeeds, the proof of success of the approximation is converted (by implication) to a proof of the property. Otherwise an error is raised.

Note that proof irrelevance of `P c` is crucial, because when going from the subset type `{c:C & P c}` to the simple type `C` and back, the element of `P c` is inferred by the approximation decision, and there is no reason for it to be the same as the initial element of `P c`. By proof-irrelevance, both proofs are considered equal, and the partial equivalence is established.

4.2.4 Equivalence Between Dependent and Simple Types

We now define a partial equivalence between dependent and simple types by composing the two equivalences described above. The *dependent equivalence* class below captures the necessary requirements for two structures to interoperate. `DepEquiv` corresponds to a *first order* dependent interoperability, in as much as it only relates data structures. In Section 4.3, we shall develop higher-order dependent equivalences, which enable us to operate over functions. As explained above, we define a class so as to piggy-back on instance resolution to find the canonical partial equivalences automatically.

```
Class DepEquiv (A : Type) (B: A → Type) (C : HSet) := {
   P : A → C → HProp;
   total_equiv :> ∀ a, B a ≃ {c:C & P a c};
   partial_equiv :> ∀ a, {c:C & P a c} ≃?K C;
   f_ca : C ⇀ A;
   P_f_ca : ∀ a (b:B a), (f_ca ∘K pek_fun) (e_fun b) = Some a;
  }.
Notation "B ≈ C" := (DepEquiv _ B C).
```

(The index type `A` can always be inferred from the context so the notation ≈ omits it.) A key ingredient to establishing a dependent equivalence between the type family `B` and the simple type `C` is the property `P` that connects the two equivalences. Note that the partial and total equivalences with the subset type are lifted to point-wise equivalences, *i.e.*, they must hold for all indices a.[9]

The `DepEquiv` class also includes an index synthesis function, $\mathtt{f}_{ca}$, which recovers a canonical index from a data of simple type. In the case of an $\mathtt{List}_{\mathbb{N}}$, it is always possible to compute its length, but as we will see in the case of stack machine instructions (Section 4.4), synthesizing an index may fail. The $\mathtt{f}_{ca}$ function is used for defining higher-order equivalences, *i.e.*, for automatically lifting functions (Section 4.3). The property $\mathtt{P}_{\mathtt{f}_{ca}}$ states that if we have a value `c : C` that was obtained through the equivalence from a value of type `B a`, then $\mathtt{f}_{ca}$ is defined on `c` and recovers the original index `a`.

Finally, for all index `a`, `B ≈ C` is a partial type equivalence in the Kleisli category of the `Cast` monad:

```
Definition DepEquiv_PEK (A : Type) (B : A → Type)
      (C : HSet) `{B ≈ C} (a:A) : B a ≃?K  C :=
 {| pek_fun := to_simpl;
    pek_isequiv := {| pek_inv := to_dep a |} |}.
```

[9] To define a dependent equivalence, Coq must also be able to infer that the type `C` is an `HSet`. In practice, it is convenient to exploit Hedberg's theorem [Uni13, Section 7.2], which states that decidable equality on `T` (which is easier to prove) implies `isHSet T`.

The functions used to establish the partial equivalence are `to_simpl`, which is the standard composition of the two equivalence functions `pek_fun ∘ e_fun`, and the function `to_dep`, which is the Kleisli composition of the inverse functions, `(clift e_inv)` $\circ_K$ `pek_inv`.

4.2.5 Simplifying the Definition of a Dependent Equivalence

In practice, requiring the simple type `C` to be an `HSet` allows to alleviate the burden on the user, because some coherences become automatically satisfied. We define a function `IsDepEquiv` that exploits this and creates a dependent equivalence without requiring the extra coherences `e_adj` or `pek_adj`.

Additionally, note that the `DepEquiv` class is independent of the particular partial equivalence between the subset type and the simple type. Therefore, we provide a smart constructor for dependent equivalences, applicable whenever the partial equivalence with the subset type is given by a checkable property:

```
Definition IsDepEquiv {A: Type} (B: A → Type) (C:HSet)
  (P: A → C → HProp) `{∀ a c, Checkable (P a c)}
   (f_bc : ∀ a, B a → {c : C & P a c})
   (f_cb : ∀ a, {c : C & P a c} → B a)
   (f_ca : C ⇀ A) :
   (∀ a, (f_cb a) ∘ (f_bc a) == id) →
   (∀ a, (f_bc a) ∘ (f_cb a) == id) →
   (∀ a (b:B a), f_ca (f_bc _ b).1 = Some a) → B ≈ C.
```

Using `IsDepEquiv`, establishing a new dependent interoperability between two types such as $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$ boils down to providing a checkable predicate, two inverse conversion functions (as in Section 4.2.5), and the index synthesis function (`length`). The programmer must then prove three equations corresponding to the properties of conversions and that of the index synthesis function.

Frequently, the checkable predicate merely states that the synthesized index is equal to the proposed index (*i.e.*, `P := fun a c ⇒ f`$_{ca}$`_eq c = Some a`). We provide another convenient instance constructor `DepEquiv_eq`, specialized to handle this situation. Declaring the canonical dependent equivalence between $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$ amounts to:

```
Instance DepEquiv_vector_list : Vec_ℕ ≈ List_ℕ :=
   DepEquiv_eq Vec_ℕ List_ℕ (clift length)
                  vector_to_list list_to_vector.
```

4.3 Higher-Order Partial Type Equivalence

Having defined first-order dependent equivalences, which relate indexed types (*e.g.*, $\texttt{Vec}_{\mathbb{N}}$) and simple types (*e.g.*, $\texttt{List}_{\mathbb{N}}$), we now turn to *higher-order* dependent equivalences, which rely on higher-order partial type equivalences. These higher-order equivalences relate partial functions over simple types, such as $\texttt{List}_{\mathbb{N}} \rightharpoonup \texttt{List}_{\mathbb{N}}$, to partially-equivalent functions over indexed types, such as $\forall$ `n.` $\texttt{Vec}_{\mathbb{N}}$ `(n + 1)` $\rightharpoonup$ $\texttt{Vec}_{\mathbb{N}}$ `n`.

Higher-order equivalences support the application scenarios of dependent interoperability described in the introduction. Importantly, while programmers are expected to define their own first-order dependent equivalences, higher-order equivalences are automatically derived based on the available canonical first-order dependent equivalences.

4.3.1 Defining A Higher-Order Dependent Interoperability

Consider that two first-order dependent equivalences $\mathtt{B_1} \approx \mathtt{C_1}$ and $\mathtt{B_2} \approx \mathtt{C_2}$ have been previously established. We can construct a higher-order partial type equivalence between functions of type $\forall$ `a:A, B1 a` $\rightharpoonup$ `B2 a` and functions of type `C1` $\rightharpoonup$ `C2`:

```
Instance HODepEquiv {A: Type}
 {B1: A → Type} {C1: HSet} `{B1 ≈ C1}
 {B2: A → Type} {C2: HSet} `{B2 ≈ C2} :
 (∀ a:A, B1 a ⇀ B2 a) ≃? (C1 ⇀ C2) :=
{| pe_fun := fun f ⇒ to_simpl_dom
                          (fun a b ⇒ x ← f a b;
                                      to_simpl x)) _ ;
   pe_isequiv := {| pe_inv :=
                       fun f a b ⇒ x ← to_dep_dom f a b;
                                    to_dep _ x) _ |}|}.
```

The definition of the `HODepEquiv` instance relies on two new auxiliary functions, `to_dep_dom` and `to_simpl_dom`.

`to_dep_dom` lifts a function of type `C` $\rightharpoonup$ `X` for any type `X` to an equivalent function of type $\forall$ `a. B a` $\rightharpoonup$ `X`. It simply precomposes the function to lift with `to_simpl` in the Kleisli category:

```
Definition to_dep_dom {A X} {B: A → Type} {C : HSet}
       `{B ≈ C} (f: C ⇀ X) (a:A) : B a ⇀ X := f ∘_K to_simpl.
```

`to_simpl_dom` lifts the domain of a function in the other direction. Its definition is more subtle because it requires computing the index `a` associated to `c` before applying `to_dep`. This is precisely the *raison d'être* of the $\mathtt{f}_{ca}$ function provided by the `DepEquiv` type class.

```
Definition to_simpl_dom {A X} {B: A → Type} {C : HSet}
       `{B ≈ C} (f : ∀ a:A, B a ⇀ X) : C ⇀ X :=
 fun c ⇒ a ← f_ca c;
          b ← to_dep a c ;
          f a b.
```

Crucially, the proof that `HODepEquiv` is a partial equivalence is done once and for all. This is an important asset for programmers because the proof is quite technical. It implies proving equalities in the Kleisli category and requires in particular the extra property $\mathtt{P}_{\mathtt{f}_{ca}}$. We come back to proof techniques in Section 4.4.6.

As the coherence condition of `HODepEquiv` involves equality between functions, the proof makes use of the functional extensionality axiom, which states that `f == g` is equivalent to `f = g` for any dependent functions `f` and `g`. This axiom is very common and compatible with both UIP and univalence, but it can not be proven in Coq for the moment, because equality is defined as an inductive type, and the dependent product is of a coinductive nature.

In order to cope with pure functions of type $\forall$ `a, B a` $\rightarrow$ `C a`, we first embed the pure function into the monadic setting and then apply a partial equivalence:

```
Definition lift {A} {B1: A → Type} {B2 : A → Type}
   {C1 C2 : HSet} `{∀ a, B1 a ⇀ B2 a ≃? C1 ⇀ C2} :
  (∀ a, B1 a → B2 a) → C1 ⇀ C2 :=
 fun f ⇒ pe_fun (fun a b ⇒ creturn (f a b)).
```

This definition is straightforward, yet it provides a convenient interface to the user of the dependent interoperability framework. For instance, lifting the function:

```
Vec_ℕ.map : ∀ (f : ℕ → ℕ) (n : ℕ), Vec_ℕ n → Vec_ℕ n
```

is a mere `lift` away:

```
Definition map_simpl (f : ℕ → ℕ) : list ℕ ⇀ list ℕ
  := lift (Vec_ℕ.map f).
```

Note that it is however not (yet) possible to lift the tail function `Vec_ℕ.tl : ∀ n, Vec_ℕ (S n) → Vec_ℕ n` because there is no dependent equivalence between `Vec_ℕ (S n)` and `List_ℕ`. Fortunately, the framework is extensible and we will see in the next section how to deal with this example, among others.

4.3.2 A Library of Higher-Order Dependent Equivalences

`HODepEquiv` is but one instance of an extensible library of higher-order dependent equivalence classes. One of the benefits of our approach to dependent interoperability is the flexibility of the framework. Automation of higher-order dependent equivalences is open-ended and user-extensible. We now discuss some useful variants, which provide a generic skeleton that can be tailored and extended to suit specific needs.

Index injections. `HODepEquiv` only covers the pointwise application of a type index over the domain and codomain types. This fails to take advantage of full-spectrum dependent types: a type-level function could perfectly be applied to the type index. For instance, if we want to lift the tail function `Vec_ℕ.tl : ∀ n, Vec_ℕ (S n) → Vec_ℕ n` to a function of type `List_ℕ → List_ℕ`, then the domain index is obtained from the index `n` by application of the successor function.

Of particular interest is the case where the index function is an inductive constructor. Indeed, inductive families are commonly defined by case analysis over some underlying inductive type [BMM04]. Semantically, we characterize constructors through their defining characteristic: they are *injective*. We thus define a class of injections where the inverse function is allowed to fail:

```
Class IsInjective {A B : Type} (f : A → B) := {
    i_inv : B ⇀ A;
    i_sect : i_inv ∘ f == creturn ;
    i_retr : clift f ∘_K i_inv ⪯ creturn
  }.
```

We can then define a general instance of `DepEquiv` that captures the use of an injection on the index. Note that for the sake of generality, the domain of the injection can be a different index type `A'` from the one taken by `B`:

```
Instance DepEquivInj (A A' : Type) (B : A → Type)
  (C : HSet) (f : A' → A) `{IsInjective f} `{B ≈ C} :
    (fun a ⇒ B (f a)) ≈ C
```

This new instance now makes it possible to lift the tail function from vectors to lists:

```
Definition pop : list ℕ ⇀ list ℕ := lift Vec_ℕ.tl.
```

As expected, when applied to the empty list, the function `pop` returns `None`, which corresponds to the error of the `Cast` monad.[10] In the other direction, we can as easily lift a `pop` function on lists to the dependent type `∀ n, Vec_ℕ (S n) ⇀ Vec_ℕ n`. This function can only be applied to a non-empty vector, but if it does not return a vector of a length reduced by one, a cast error is reported.

[10] We come back to an improvement of the error message in Section 4.4.5

Composing equivalences. With curried dependently-typed functions, the index of an argument can be used as an index of a subsequent argument (and return type), for instance:

```
∀ a a', B1 a → B2 a a' ⇀ B3 a a'
```

We can define an instance of $\simeq^?$ to form a new partial equivalence on `∀ a a', B1 a → B2 a a' ⇀ B3 a a'` from a partial equivalence on `∀ a', B2 a a' ⇀ B3 a a'`, for a fixed `a`, provided that we have established that $B_1 \approx C_1$:

```
Instance HODepEquiv2 A A' B1 B2 B3 C1 C2 C3
  `{∀ a, ((∀ a':A', B2 a a' ⇀ B3 a a') ≃? (C2 ⇀ C3))}
  `{B1 ≈ C1}:
 (∀ a a', B1 a → B2 a a' ⇀ B3 a a') ≃? (C1 → C2 ⇀ C3).
```

For space reasons, we do not dive into the technical details of this instance, but it is crucial to handle the stack machine example of the introduction: as explained in Section 4.4, the example involves composing two dependent equivalences, one on instructions and one on stacks.

Index dependencies. It is sometimes necessary to reorder arguments in order to be able to compose equivalences, accounting for (functional) dependencies between indices. This reordering of parameters can be automatized by defining an instance that tries to flip arguments to find a potential partial equivalence:

```
Instance HODepEquiv2_sym A A' B1 B2 B3 C1 C2 C3
`{∀ a a', B2 a a' → B1 a a' ⇀ B3 a a') ≃? (C2 → C1 ⇀ C3)} :
 (∀ a a', B1 a a' → B2 a a' ⇀ B3 a a') ≃? (C1 → C2 ⇀ C3)
```

Note that this instance has to be given a very low priority (omitted here) because it must be used as a last resort, or one would introduce cycles during type class resolution. The stack machine example in Section 4.4 also exploits this instance.

4.4 A Certified, Interoperable Stack Machine

To demonstrate our approach, we address the shortcomings of extraction identified in Section 4 and present a certified yet interoperable interpreter for a toy stack machine. Let us recall the specification of the interpreter:

```
exec: ∀ n m, dinstr n m → dstack n → dstack m
```

This definition enforces—by construction—an invariant relating the size of the input stack and output stack, based on which instruction is to be executed.

In the simply-typed setting, we would like to offer the following interface:

```
safe_exec: instr → ListN → ListN
```

while dynamically enforcing the same invariants (and rejecting ill-formed inputs).

This example touches upon two challenges. First, it involves two equivalences, one dealing with instructions and the other dealing with stacks. Once those equivalences have been defined by the user, we shall make sure that our machinery automatically finds them to lift the function `exec`. Second, and more importantly, type indices flow in a non-trivial manner through the type signature of `exec`. For instance, providing an empty stack means that we must forbid the use of the `IPlus` instruction. Put otherwise, the lifting of the dependent instruction depends on the (successful) lifting of the dependent stack. As we

shall see, the index `n` is (uniquely) determined by the input list size while the index `m` is (uniquely) determined by `n` and the instruction being executed. Thus, the automation of higher-order lifting needs to be able to linearize such indexing flow and turning them into sequential checks.

In this process, users are only asked to provide the two first-order type equivalences specific to their target domain, `dstack` $\approx$ `List`$_{\mathbb{N}}$ and $\forall$ `n, dinstr n` $\approx$ `instr`. Using these instances, the role of our framework is threefold: (1) to linearize the indexing flow, through potentially reordering function arguments; (2) to identify the suitable first-order equivalences, through user and library provided instances; (3) to propagate the indices computed through dynamic checks, through the constructive reading of the higher-order equivalences (Section 4.3).

4.4.1 From Stacks to Lists

As hinted at in Section 4, the type of `dstack` cannot be properly extracted to a simply-typed system. Indeed, it is defined by large elimination over natural numbers and there is therefore no natural, simply-typed data structure to extract it to. As a result, extraction in Coq resorts to unsafe type coercions [Let04, Section 3.2]. However, using specific domain knowledge, the programmer can craft an equivalence with a list, along the lines of the equivalence between vectors and lists. We therefore (constructively) witness the following subset equivalence:

```
dstack n ≃ {l : List_ℕ & clift length l = Some n}
```

by which we map size-indexed tuples to lists.[11] Crucially, this transformation involves a change of representation: we move from tuples to lists. For our framework to automatically handle this transformation, we declare the suitable instance of dependent equivalence:

```
Instance DepEquiv_dstack : dstack ≈ List_ℕ :=
      DepEquiv_eq dstack List_ℕ  (clift length)
                      dstack_to_list list_to_dstack.
```

The definition of this dependent equivalence is very similar in nature to the one already described between vectors and lists, so we refer the reader to the implementation for details.

4.4.2 From Indexed Instructions to Simple Instructions

The interoperable version of indexed instructions is more natural to construct: indexed instructions are a standard inductive family whose indices play a purely logical role.

```
Inductive dinstr: ℕ → ℕ → Set :=
| IConst: ∀ n, ℕ → dinstr n (S n)
| IPlus: ∀ n, dinstr (S (S n)) (S n).
```

Merely erasing this information gives an inductive type of (simple) instructions:

```
Inductive instr : Type :=
| NConst : ℕ → instr
| NPlus  : instr.
```

[11] Note that the equality `clift length l = Some n` is equivalent to the simpler `length l = n`, but the framework is tailored to encompass potential failure. This could be avoided by defining a more specific function than `DepEquiv_eq` for the case where computation of the index never fails.

Nonetheless, relating the indexed and simple version is conceptually more involved. Indeed, the first index cannot be guessed from the simply-typed representation alone: the size of the input stack must be provided by some other means. Knowing the size of the input stack, we can determine the expected size of the output stack for a given simple instruction:

```
Definition instr_index n (i:instr) : Cast ℕ :=
 match i with
  | NConst _ ⇒ Some (S n)
  | NPlus ⇒ match n with
             | S (S n) ⇒ Some (S n)
             | _ ⇒ None
            end
 end.
```

The dependent equivalence is thus *parameterized* by the input size `n` and only the output size `m` needs to be determined from the simple instruction:

```
∀ n, dinstr n m ≃ {i: instr &  instr_index n i = Some m}.
```

Once again, we inform the system of this new equivalence through a suitable instance declaration:

```
Instance DepEquiv_instr n : (dinstr n) ≈ instr :=
 DepEquiv_eq (dinstr n) instr (instr_index n)
                 (dinstr_to_instr n) (instr_to_dinstr n).
```

4.4.3 Lifting the Interpreter

Having specified our domain-specific equivalences, we are left to initiate the instance resolution so as to *automatically* obtain the desired, partially-equivalent lifting of the interpreter `exec`. To do so, we simply appeal to the `lift2` operator, which is similar to `lift` from Section 4.3.1 save for the fact that it deals with two-index types:

```
lift2 A A' B1 B2 B3 C1 C2 C3
 `{∀ a a', B1 a a' → B2 a a' ⇀ B3 a a' ≃? C1 → C2 ⇀ C3} :
  (∀ a a', B1 a a' → B2 a a' → B3 a a') → C1 → C2 ⇀ C3.
```

The definition of `simple_exec` is then:

```
Definition simple_exec : instr → List_ℕ ⇀ List_ℕ :=
  lift2 exec.
```

`lift2` matches upon the skeleton of our dependent function `exec`, lifts it to a monadic setting and triggers the instance resolution mechanism of Coq. This (single) command is enough to build the term `simple_exec` with the desired type, together with the formal guarantee that it is partially-equivalent to the dependent program we started from.

4.4.4 Diving into the Generated Term

Printing the generated term (by telling Coq to show partial equivalence instances) is instructive:

```
simple_exec = lift2
 (HODepEquiv2_sym
```

```
  (HODepEquiv2
    (fun a : ℕ ⇒ HODepEquiv (DepEquiv_instr a)
                            DepEquiv_stack)
    DepEquiv_stack)) exec
```

We witness three generic transformations of the function: `HODepEquiv2_sym`, which has reordered the input arguments so as to first determine the size of the input stack; `HODepEquiv2`, which has made the size of the input list available to subsequent transformations; and `HODepEquiv`, which has transferred the size of the output list as computed from the simple instruction to the output stack.

Now, when printing `simple_exec` by telling Coq to unfold definitions, we recover the description of a function in monadic style that actually performs the expected computation:

```
simple_exec =  fun (i : instr) (l : List_ℕ) ⇒
(* lift l to a dstack ds of size (length l) *)
  ds ← (c' ← to_subset l; Some (list_to_dstack c'));
(* compute the index associated to (length l) for i
   this may fail depending on the instruction *)
  m ← instr_index (length l) i;
(* lift i to a dependent instruction di *)
  di ← (c' ← to_subset i;
        Some (instr_to_dinstr (length l) m c'));
(* perform exec (note the reverse order of di and ds)
   and convert the result to a list *)
  Some (dstack_to_list (exec (length l) m di ds)) .1
```

4.4.5 Extraction to OCaml

In Section 4.1.2, we introduced the `Cast` monad as *essentially* the option monad. For practical purposes, the failure constructor of `Cast` takes additional arguments for producing informative error messages: we capture the type we are trying to cast to and a message to help diagnose the source of the error.

More importantly, having defined a custom error monad enables us to tailor program extraction when targeting an impure language. In an impure language like OCaml, it is indeed possible—and strongly advised—to write in direct style, using runtime exceptions to implement the `Cast` monad. The success constructor of the monad is simply erased, and its failure constructor is projected to a runtime exception (*e.g.*, `failwith` in OCaml). This allows us to avoid affecting the consistency of the host language Coq —conversely to [TT15], we do not introduce inconsistent axioms to represent cast errors— while preserving the software engineering benefits of not imposing a monadic framework on external components. The definition of the extraction of the `Cast` monad is provided in Appendix 4.6.

We can now revisit the interaction with the extracted function:

```
# simple_exec NPlus [1;2];;
- : int list = [3]
# simple_exec NPlus [];;
Exception: (Failure "Cast failure: invalid instruction").
```

and confirm that an invalid application of `simple_exec` does not yield a segmentation fault, but an informative exception.

4.4.6 A Homotopical Detour

Before concluding, we briefly reflect on the proof techniques we used to build the verified dependent interoperability framework and implement the different examples.

Many of the proofs of sections and retractions, either on general instances (Sections 4.2 and 4.3) or on domain-specific equivalences (as we shall see below), require complex reasoning on equality. This means that particular attention must be paid to the definition of conversion functions. In particular, the manipulation of equality must be done through explicit rewriting using the transport map (which is the predicative version of `ap` introduced in Section 4.1):

```
Definition transport {A : Type} (P : A → Type) {x y : A}
 (p : x = y) : P x → P y.
Notation "p # x" := (transport _ p x).
```

Transport is trivially implemented by path induction, but making explicit use of transport is one of the most important technical insights brought by the advent of HoTT. It is crucial as it enables to encapsulate and reason abstractly on rewriting, without fighting against dependent types.[12] Indeed, although equality is *presented* through an inductive type in Coq, it remains best dealt with through abstract rewriting— a lesson that was already familiar to observational type theorists [AMS07]. The reason is that it is extremely difficult to prove equality of two pattern matching definitions by solely reasoning by pattern matching. Conversely, it is perfectly manageable to prove equality of two different transportations.

For instance, the definition of `instr_to_dinstr` must be defined by pattern matching on the instruction and transport (comments express the specific type of the goal in each branch of pattern matching):

```
Definition instr_to_dinstr n n' :
 {i: instr & instr_index n i = Some m} → dinstr n n' := fun x ⇒
match x with (i;v) ⇒ (match i with
 (* ⊢ Some (S n) = Some n' → dinstr n n' *)
 | NConst k ⇒ fun v ⇒ Some_inj v # IConst k
 | NPlus    ⇒ match n with
 (* ⊢ None = Some n' → dinstr 0 n' *)
    0 ⇒ fun v ⇒ None_is_not_Some v
 (* ⊢ None = Some n' → dinstr 1 n' *)
   | S 0 ⇒ fun v ⇒ None_is_not_Some v
 (* ⊢ Some (S n) = Some n' → dinstr (S (S n)) n' *)
   | S (S n) ⇒ fun v ⇒ Some_inj v # IPlus
end end) v end.
```

where `None_is_not_Some` is a proof that `None` is different from `Some a` for any `a` (in the sense that `None = Some a` implies anything) and `Some_inj` is a proof of injectivity of the constructor `Some`.

The benefit of using the encapsulation of rewriting through transport is that now, we can prove auxiliary lemmas on transport and use them in the proof. For instance, we can state how transport behaves on the `IConst` instruction by path induction:

```
Definition transport_instr_Const (n m k : ℕ) (e : S n = m) :
dinstr_to_instr (e # (IConst n0)) = (NConst n0; ap Some e).
```

[12] A fight that Coq usually announces with "`Abstracting over the terms...`" and wins by declaring "`is ill-typed.`"

and similarly for IPlus. Armed with these properties on transport, we can then prove the retraction of dinstr m n ≃ {i: instr & instr_index n i = Some m} by pattern matching on instructions and integers, together with some groupoid laws (@ is equality concatenation and path_sigma is a proof that equalities of the projections imply equality dependent pairs).

```
Definition DepEquiv_instr_retr n m
              (x:{i:instr & instr_index n i = Some m}) :
 (dinstr_to_instr n m) ∘ (instr_to_dinstr n m) x = x :=
 match x with (i;v) ⇒ (match i with
(*  ⊢  Some (S n) = Some m →
      dinstr_to_instr n m (Some_inj v # IConst n0)
          = (NConst n0; v) *)
  NConst k ⇒ fun v ⇒
         transport_instr_Const @
         path_sigma eq_refl (is_hprop _ _)
| NPlus ⇒ match n with
(*  ⊢  None = Some m →
      dinstr_to_instr 0 m (Fail_is_not_Some v)
          = (Nplus; v) *)
        0 ⇒ fun v ⇒ None_is_not_Some v
(*  ⊢  None = Some m →
      dinstr_to_instr 1 m (Fail_is_not_Some v)
          = (Nplus; v) *)
       | S 0 ⇒ fun v ⇒ None_is_not_Some v
(*  ⊢  Some (S n) = Some m →
      dinstr_to_instr (S (S n)) m (Some_inj v # IPlus)
          = (NPlus; v) *)
       | S (S n) ⇒ fun v ⇒
         transport_instr_Plus @
         path_sigma eq_refl (is_hprop _ _)) end
        end) v end end.
```

We believe that this new way of proving equalities—initially introduced to manage higher equalities in syntactical homotopy theory—is very promising for proving equalities on definitions done by pattern matching and thus proving properties on dependent types.

4.5 Related Work

As far as we know, the term *dependent interoperability* was originally coined by [OSZ12] as a particularly challenging case of *multi-language semantics* between a dependently-typed and a simply-typed language. The concept of multi-language semantics was initially introduced by [MF07] to capture the interactions between a simply-typed calculus and a uni-typed calculus (where all closed terms have the same unique type).

Our approach is strictly more general in that we make no assumption on the dependent types we support: as long as the user provides partial type equivalences, our framework is able to exploit them automatically. In particular, we do not require a one-to-one correspondence between constructors: the equivalence is established at the type level, giving the user the freedom to implement potentially partial transformations. We also account for more general equivalences through partial index synthesis functions; [OSZ12] assume that these functions are total and manually introduced by users. Finally, while

their work is fundamentally grounded in a syntactic treatment of interoperability, ours takes its roots in a semantic treatment of type equivalences internalized in Coq. We are thus able to give a presentation from first principles while providing an executable toolbox in the form of a Coq library that is entirely verified.

Dynamic typing with dependent types. Dependent interoperability can also be considered within a single language, as explored by [OTMW04]. The authors developed a core language with dependent function types and subset types augmented with three special commands: `simple{e}`, to denote that expression `e` is simply well-typed, `dependent{e}`, to denote that the type checker should statically check all dependent constraints in `e`, and `assert(e, T)` to check at runtime that `e` produces a value of (possibly-dependent) type `T`. The semantics of the source language is given by translation to an internal language relying, when needed, on runtime-checked type coercions.

However, dependent types are restricted to refinement types where the refinements are pure Boolean expressions, as in [KF10]). This means that the authors do not address the issues related to indexed types, including that of providing correct marshalling functions between representations, which is a core challenge of dependent interoperability.

Casts for subset types. [TT15] also explore the interaction between simple types and refinements types in a richer setting than [OTMW04] : their approach is developed in Coq, and thus refinements are any proposition (not just Boolean procedures), and they accommodate explicitly proven propositions. They support sound casts between simple types and subset types by embedding runtime checks to ensure that the logical component of a subset type is satisfied. Our notion of dependent equivalence builds upon the idea of casting to subset types—we use subset types as mediators between simple types and indexed types. But instead of using an inconsistent axiom in the computational fragment of the framework to represent cast errors, we operate within a `Cast` monad (recall that we do use a fairly standard axiom, functional extensionality, in the non-computational fragment of the framework). Imposing a monadic style augments the cost of using our framework within Coq, but we can recover the convenience of non-monadic signatures upon extraction. Finally, just like [OTMW04], the work of [TT15] only deals with subset types and hence does not touch upon dependent interoperability in its generality.

The fact that dependent equivalences only abstractly rely on casting to subset types should make it possible to derive instances for other predicates than the `Checkable` ones. For instance, in the setting of the Mathematical Components library using the SSreflect proof language, properties are better described through Boolean reflection [GM10]. Using Boolean functions is very similar to using decidable/checkable properties, so that framework should provide a lot of new interesting instances of partial equivalences between subset and simple types in the Kleisli category of the `Cast` monad.

Gradual typing. Multi-language semantics are directly related to *gradual typing* [ST06], generalized to denote the integration of type disciplines of different strengths within the same language. This relativistic view of gradual typing has already been explored in the literature for disciplines like effect typing [BGT14] and security typing [DF11, FT13]. Compared to previous work on gradual typing, dependent interoperability is challenging because the properties captured by type indices can be semantically complex. Also, because indices are strongly tied to specific constructors, casting requires marshalling [OSZ12].

In our framework, casting (which we termed "lifting") must be explicitly summoned. However, as already observed by [TT15], the implicit coercions of Coq can be used to

direct the type checker to automatically insert liftings when necessary, thus yielding a controlled form of gradual typing.

This being said, there is still work to be done to develop a full theory of gradual dependent types. It would be interesting to explore how the abstract interpretation foundations of gradual typing as a theory of dealing with type information of different levels of precision [GCT16] can be connected with our framework for dependent equivalences, which relate types of different precision.

Ornaments. Our work is rooted in a strict separation of the computational and logical content of types, which resonates with the theory of ornaments [McB10], whose motto is "datatype = data structure + data logic". Ornaments have been designed as a construction kit for inductive families: while data structures—concrete representation over which computations are performed—are fairly generic, data-logics—which enforce program invariants—are extremely domain-specific and ought to be obtained on-the-fly, from an algebraic specification of the invariants.

In particular, two key ingredients of the ornamental toolbox are algebraic ornaments [McB10] and relational ornaments [KG13]. From an inductive type and an algebra (over an endofunctor on $\mathcal{S}et$ for algebraic ornaments, over an endofunctor on $\mathcal{R}el$ for relational ornaments), these ornaments construct an inductive family satisfying—by construction—the desired invariants. The validity of this construction is established through a type equivalence, which asserts that the inductive family is equivalent to the subset of the underlying type satisfying the algebraic predicate.

However, the present work differs from ornaments in several, significant ways. First, from a methodological standpoint: ornaments aim at creating a combinatorial toolbox for creating dependent types from simple ones. Following this design goal, ornaments provide correct-by-construction transformation of data structures: from a type and an algebra, one obtains a type family. In our framework, both the underlying type and the indexed type must pre-exist, we merely ask for a (constructive) proof of type equivalence. Conceptually, partial type equivalences subsume ornaments in the sense that an ornament automatically gives rise to a partial type equivalence. However, ornaments are restricted to inductive types, while partial type equivalences offer a uniform framework to characterize the refinement of any type, including inductive families.

Functional ornaments. To remediate these limitations, the notion of functional ornaments [DM12] was developed. As for ornaments, functional ornaments aim at transporting functions from simple, non-indexed types to more precise, indexed types. The canonical example consists in taking addition over natural numbers to concatenation of lists: both operations are strikingly similar and can indeed be described through ornamentation. Functional ornaments can thus be seen as a generalization of ornaments to first-order functions over inductive types.

So far, however, such generalization of ornaments have failed to carry over higher-order functions and genuinely support non-inductive types. The original treatment of functional ornaments followed a semantic approach, restricting functions to be catamorphisms and operating over their defining algebras. More recent treatment [WDR14], on the other hand, is strongly grounded in the syntax, leading to heuristics that are difficult to formally rationalize.

By focusing on type equivalences, our approach is conceptually simpler: our role is to consistently preserve type information, while functional ornaments must infer or create well-indexed types out of thin air. By restricting ourselves to checkable properties, we also afford the flexibility of runtime checks and, therefore, we can simply lift values to and

from dependent types by merely converting between data representations. Finally, while the original work on functional ornaments used a reflective universe, we use type classes as an open-ended and extensible meta-programming framework. In particular, users are able to extend the framework at will, unlike the clients of a fixed reflective universe.

Refinement types. Our work shares some similarities with refinement types [RKJ08]. Indeed, dependent equivalences are established through an intermediary type equivalence with user-provided subset types, which is but a type-theoretic incarnation of a refinement type. From refinement types, we follow the intuition that most program invariants can be attached to their underlying data structures. We thus take advantage of the relationship between simple and indexed types to generate runtime checks. Unlike [SNI15], our current prototype fails to take advantage of the algebraic nature of some predicates, thus leading to potentially inefficient runtime checks. In principle, this shortcoming could be addressed by integrating algebraic ornaments in the definition of type equivalences. Besides, instead of introducing another manifest contract calculus and painstakenly developing its meta-theory, we leverage full-spectrum dependent types to internalize the cast machinery through partial type equivalences.

Such internalization of refinement techniques has permeated the interactive theorem proving community at large, with applications ranging from improving the efficiency of small-scale reflection [CDM13], or the step-wise refinement of specifications down to correct-by-construction programs [DPGC15, SA16]. Our work differs from the former application because we are interested in *safe* execution outside of Coq rather than *fast* execution in the Coq reduction engine. It would nonetheless be interesting to attempt a similar parametric interpretation of our dependent equivalences. Our work also differs from step-wise refinements in the sense that we transform dependently-typed programs to and from simply-typed ones, while step-wise refinements are concerned with incrementally determining a relational specification.

4.6 Future Work

As a first step, we wish to optimize the runtime checks compiled into the interoperability wrappers. Indeed, dependent types often exhibit a tightly-coupled flow of indexing information. Case in point is the certified stack machine, whose length of the input list gives away the first index of the typed instruction set while its second index is obtained from the raw instruction and the input length. By being systematic in our treatment of such dataflows, we hope to identify their sequential treatments and thus minimize the number of (re)computations from simply-typed values.

The similarities with standard dataflow analysis techniques are striking. Some equivalences (typically, `DepEquiv_eq`) are genuine *definition sites*. For instance, the input list of the stack machine *defines* its associated index. Other equivalences are mere *use sites*. For instance, the first argument of the typed instruction cannot be determined from a raw instruction: one must obtain it from the input list. As hinted at earlier, the second argument of the typed instruction can be computed from the first one, thus witnessing a *use-definition chain*. Conceptually, lifting a dependently-typed program consists in performing a topological sort on this dataflow graph.

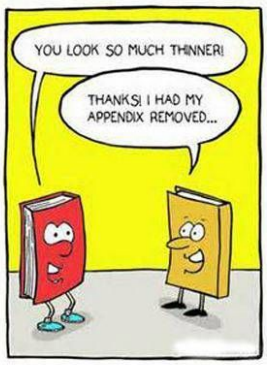
Taking full advantage of this representation opens many avenues for generalizations. For instance, our current definition of dependent equivalences insists on being able to recover an index from a raw value through the mandatory $\texttt{f}_{ca}$ `: C` $\rightharpoonup$ `A`. As such, this precludes the definition of many equivalences, such as the equivalence between natural

numbers and finite sets (*i.e.*, bounded natural numbers, the bound being unknown), or between raw lambda terms and intrinsic dependently-typed terms (the types being unknown and, *a priori*, not inferable).

Finally, perhaps inspired by the theory of ornaments, we would like to avoid marshalling values across inductive types and their algebraically ornamented families. Indeed, when converting from, say, lists to vectors, we perform a full traversal of the list to convert it to a vector that is, essentially, the same datatype. Most of our conversion functions are nothing but elaborate identity functions. By taking advantage of this structural information and, perhaps, some knowledge of the extraction mechanism, we would like to remove this useless and inefficient indirection.

Coq Formalization. The full Coq formalization, with documentation, is available at `http://coqhott.github.io/DICoq/`. It has been developed using the 8.5 release of Coq [CDT15].

Appendix



-You look so much thinner.
-Thanks, I had my appendix removed.

Type Theory

CC_ω is a type theory featuring only negative connectives, i.e. Π-types. It features a denumerable hierarchy of universes $\mathbf{Type}_i$ together with an impredicative universe $*$, and is therefore essentially Luo's ECC without pairs nor cumulativity [Luo89].

Definition 46 (*Typing system*)

As usual, we define here two statements mutually recursively. The statement $\vdash \Gamma$ *means that the environment* Γ *is well-founded, while* $\Gamma \vdash M : A$ *means that the term* M *has type* A *in environment* Γ*. We write* $\mathbf{Type}$ *for* $*$ *or* $\mathbf{Type}_i$ *for some* $i \in \mathbb{N}$*. The typing rules are given at Figure 4.1. The rules are almost entirely standard. The definitional equality* $A \equiv B$ *is defined as the congruence on type and term formers compatible with* β*-reductions for abstractions and projections. We give a version of the theory with named variables for presentation purposes but the interpretation is actually defined on the explicit substitution version of the system, without the xi rule allowing to push substitutions under abstractions.*

Sigma type. CC_ω is extended to CIC by adding inductive types in the theory. For simplicity, we present the definition of Σ types (Fig 4.2), which corresponds to dependent products.

Identity type. The identity type in the source theory is the standard Martin-Löf identity type $\mathtt{Id}_T\, t\, u$, generated from an introduction rule for reflexivity with the usual `J` eliminator and its *propositional* reduction rule. The `J` reduction rule will actually be valid definitionally in the model for *closed* terms only, as it relies on the potentially abstract functorial action of the elimination predicate, as in Hofmann & Streicher's interpretation.

Note that we use a slightly different version from the identity type defined in Coq as it inhabits $\mathbf{Type}_i$ for some i and not $*$. This is in accordance with the new point of view on equality given by HoTT.

$$A, B, M, N ::= * \mid \mathbf{Type}_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

$$\frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \mathbf{Type}_i : \mathbf{Type}_j} \qquad \frac{\vdash \Gamma}{\Gamma \vdash * : \mathbf{Type}_i} \qquad \frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_j}{\Gamma \vdash \Pi x : A.\, B : \mathbf{Type}_{\max(i,j)}}$$

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x : A.\, B : *}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : \mathbf{Type}}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x := N\}}$$

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \mathbf{Type}}{\Gamma, x : A \vdash M : B} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \mathbf{Type} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A : \mathbf{Type}_i}{\Gamma, x : A \vdash x : A}$$

$$\frac{}{\Gamma \vdash (\lambda x : A.\, M)\ N \equiv M\{x := N\}} \qquad \frac{\Gamma \vdash M : \Pi x : A.\, B}{\Gamma \vdash M \equiv \lambda x : A.\, M\ x}$$

(congruence rules ommitted)

Figure 4.1: Typing rules of CC_ω

$$A, B, M, N ::= \ldots \mid \Sigma x : A.\, B \mid (M, N) \mid \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N$$

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_j}{\Gamma \vdash \Sigma x : A.\, B : \mathbf{Type}_{\max(i,j)}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B\{x := M\} \qquad \Gamma \vdash \Sigma x : A.\, B : \mathbf{Type}}{\Gamma \vdash (M, N) : \Sigma x : A.\, B}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma \vdash C : \mathbf{Type} \qquad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : C}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma, z : \Sigma x : A.\, B \vdash C : \mathbf{Type} \qquad \Gamma, x : A, y : B \vdash N : C\{z := (x, y)\}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow C\{z := (x, y)\}}$$

Figure 4.2: Typing rules for Σ-types in CIC^-

The proof assistant

In this book, we use the latest version (8.5) of the Coq proof assistant. Vanilla features of Coq allow us to define overloaded notations and hierarchies of structures through type classes [SO08], and to separate definitions and proofs using the Program extension

$$\textsc{Id}\ \frac{\Gamma \vdash T : \mathbf{Type} \quad \Gamma \vdash a, b : T}{\Gamma \vdash \mathtt{Id}_T\ a\ b : \mathbf{Type}} \qquad \textsc{Id-Intro}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathtt{refl}_T\ t : \mathtt{Id}_T\ t\ t}$$

$$\textsc{Id-Elim (J)}\ \frac{\Gamma \vdash i : \mathtt{Id}_T\ t\ u \qquad \Gamma, x : T, e : \mathtt{Id}_T\ t\ x \vdash P : \mathbf{Type} \qquad \Gamma \vdash p : P\{t/x, \mathtt{refl}_T\ t/e\}}{\Gamma \vdash \mathtt{J}_{\lambda x\ e.P}\ i\ p : P\{u/x, i/e\}}$$

Figure 4.3: Typing judgments for identity type.

[Soz07], they are both documented in Coq's reference manual [CDT15]. We also use the recent extension to polymorphic universes [ST14].

Decidable and Checkable

A proposition `A` is an instance of the `Decidable` type class when there exists a function that return either a proof of `A` or a proof of `not A`. We restrict the use of the `Decidable` type class to `HProp` to force the element computed by the decidability function to be irrelevant.

```
Class Decidable (A : HProp) := dec : A + (not A).
```

A proposition is `Checkable` when there exists a decidable proposition `checkP` that implies it.

```
Class Checkable (A : HProp) := {
  checkP : HProp;
  checkP_dec : Decidable checkP ;
  convert : checkP → A;
  is_hP_c :> IsHProp A }.
```

Cast Monad

The `Cast` Monad is a refinement of the Maybe monad, that allows to collect information on the error.

```
Inductive _Cast A info :=
 | Some : A → _Cast A info
 | Fail : info → _Cast A info.
```

Here, we want to collect an error message in the form of a `string`. However, we need this extra piece of information to be irrelevant. For that, we use the *propositional truncation* `Trunc`, as introduced by Awodey and Bauer [AB04]—but in the form defined in the HoTT book [Uni13]. This allows us to state formally that the error message is irrelevant while preserving consistency and compatibility with univalence.

```
Definition Cast A := _Cast A (Trunc string).
```

We have standard monadic functions and notations:

```
Definition clift A B : (A → B) → A → Cast B :=
 fun f a ⇒ Some (f a).
```

```
Definition cbind A B : (A → Cast B) → Cast A → Cast B :=
fun f a ⇒
 match a with
    Some a ⇒ f a
  | Fail _ s t ⇒ Fail _ s t
 end.

Notation "x ← e1; e2" := cbind (fun x ⇒ e2) e1.
```

We use extraction to provide a direct style extraction of the `Cast` monad in OCaml, using runtime exceptions. The success constructor of the monad is simply erased, and its failure constructor is projected to a runtime exception where argument of the failure constructor are used as the error message.

```
(* Transparent extraction of Cast:
   - if Some t, then extract plain t
   - if Fail, then fail with a runtime cast exception *)

Extract Inductive Cast ⇒
   "" [ "" "(let f s = failwith
     (String.concat """" ([""Cast failure: ""]@
     (List.map (String.make 1) s))) in  f)"]
   "(let new_pattern some none = some in
        new_pattern)".
```

Publications of the author

Il Bibliotecario, Arcimboldo
(1556)

[BT09] Nick Benton and Nicolas Tabareau. Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In *Types in Language Design and Implementation*, Savannah, United States, January 2009.

[DTT16] Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. Partial Type Equivalences for Verified Dependent Interoperability. In *21st ACM SIGPLAN International Conference on Functional Programming*, Nara, Japan, September 2016.

[FSTT14] Ismael Figueroa, Tom Schrijvers, Nicolas Tabareau, and Éric Tanter. Compositional Reasoning About Aspect Interference. In *13th International Conference on Modularity (Modularity'14)*, Lugano, Switzerland, April 2014.

[FTT14a] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. Effect Capabilities For Haskell. In *Brazilian Symposium on Programming Languages (SBLP)*, Maceio, Brazil, September 2014.

[FTT14b] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice. *Transactions on Aspect-Oriented Software Development*, 2014.

[FTT15] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. Effect capabilities for Haskell: Taming effect interference in monadic programming. *Science of Computer Programming*, November 2015.

[HHT15] André Hirschowitz, Tom Hirschowitz, and Nicolas Tabareau. Wild omega-Categories for the Homotopy Hypothesis in Type Theory. *Leibniz International Proceedings in Informatics (LIPIcs)*, 38:226–240, 2015.

[JLP+16] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The Definitional Side of the Forcing. In *LICS*, New York, United States, May 2016.

[JTS12] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. Extending type theory with forcing. In *Logic in Computer Science (LICS), 2012*, pages 395–404. IEEE, 2012.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *Interactive Theorem Proving*, 2014.

[Tab10] Nicolas Tabareau. A theory of distributed aspects. In ACM, editor, *9th International Conference on Aspect-Oriented Software Development (AOSD '10)*, pages 133–144, Rennes, Saint-Malo, France, March 2010.

[Tab11] Nicolas Tabareau. Aspect Oriented Programming: a language for 2-categories. Research Report RR-7527, INRIA, February 2011.

[Tab12] Nicolas Tabareau. A Monadic Interpretation of Execution Levels and Exceptions for AOP. In *Modularity: AOSD'12*, Postdam, Germany, March 2012. ACM Press.

[TFT13] Nicolas Tabareau, Ismael Figueroa, and Éric Tanter. A Typed Monadic Embedding of Aspects. In *12th annual international conference on Aspect-Oriented Software Development (Modularity-AOSD'13)*, Fukuoka, Japan, March 2013.

[TT15] Éric Tanter and Nicolas Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, pages 26–40, Pittsburgh, PA, USA, October 2015. ACM Press.

Bibliography

Il Bibliotecario, Arcimboldo
(1556)

[AB04] Steven Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

[ACH+06] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[AMLH14] Carlo Angiuli, Edward Morehouse, Daniel R Licata, and Robert Harper. Homotopical patch theory. In *ICFP 2014*, volume 49, pages 243–256. ACM, 2014.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the ACM Workshop on Programming Languages meets Program Verification (PLPV 2007)*, pages 57–68, Freiburg, Germany, October 2007.

[AOS10] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press.

[AR12] Thorsten Altenkirch and Ondrej Rypacek. A Syntactical Approach to Weak omega-Groupoids. In *Computer Science Logic (CSL'12)*, volume 16, pages 16–30, 2012.

[BC15] Marc Bezem and Thierry Coquand. A kripke model for simplicial sets. *Theor. Comput. Sci.*, 574(C):86–91, April 2015.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, September 2004.

[BCH13] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *Preprint, September*, 2013.

[BCP03] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13:261–293, 3 2003.

[BCP15] Marc Bezem, Thierry Coquand, and Erik Parmann. Non-Constructivity in Kan Simplicial Sets. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 92–106, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bén67] Jean Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77. Springer-Verlag, 1967.

[BGT14] Felipe Bañados, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, September 2014. ACM Press.

[BL11] Jean-Philippe Bernardy and Marc Lasson. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, volume 6604, pages 108–122, Saarbrücken, Germany, March 2011.

[BMM04] Edwin Brady, Conor McBride, and James McKinna. *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, chapter Inductive Families Need Not Store Their Indices, pages 115–129. Springer-Verlag, 2004.

[BNSV+06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany, March 2006. ACM Press.

[BNSVV06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, and Bart Verheecke. Modularization of distributed web services using AWED. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'06*, volume 4276 of *Lecture Notes in Computer Science*, pages 1449–1466. Springer-Verlag, October 2006.

[Bru14] Aloïs Brunel. *Transformations de «forcing» et algèbres de «monitoring».* PhD thesis, 2014.

[BS06] Eric Bodden and Volker Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 147–162, Vienna, Austria, March 2006. Springer-Verlag.

[BTI14] Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 23(1):7:1–7:41, February 2014.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2016. To appear in post-proceedings of Types for Proofs and Programs (TYPES 2015).

[CDB03] Michele Curti, Pierpaolo Degano, and Cosima Tatiana Baldari. Causal π-calculus for biochemical modelling. In *Computational Methods in Systems Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 21–34. Springer-Verlag, February 2003.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Proceedings of the 3rd International Conference on Certified Programs and Proofs (CPP 2013)*, pages 147–162, Melbourne, Australia, December 2013.

[CDT15] The Coq Development Team. *The* Coq *proof assistant reference manual*. 2015. Version 8.5.

[CGH14] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theoretical Computer Science*, 546:99–119, 2014.

[Che12] Hsing-Yu Chen. COCA: Computation offload to clouds using AOP. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 466–473, Ottawa, ON, USA, 2012.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

[CL06] Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:312–374, 2006.

[CL12] Eugenia Cheng and Tom Leinster. Weak ω-categories via terminal coalgebras. *arXiv preprint arXiv:1212.5853*, 2012.

[Coh66] Paul J. Cohen. *Set theory and the continuum hypothesis*. 1966.

[DF11] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[DFES10] Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: the A calculus. In Theo D'Hondt, editor, *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, number 6183 in Lecture Notes in Computer Science, pages 101–125, Maribor, Slovenia, June 2010. Springer-Verlag.

[DLS+04] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In Lieberherr [Lie04], pages 66–73.

[DM12] Pierre-Évariste Dagand and Connor McBride. Transporting functions across ornaments. In *Proceedings of the 17th ACM SIGPLAN Conference on Functional Programming (ICFP 2012)*, pages 103–114, Copenhagen, Denmark, September 2012. ACM Press.

[DPGC15] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 689–700, Mumbai, India, January 2015. ACM Press.

[DTK06] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, September 2002. ACM Press.

[FG96a] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *23th*, pages 372–385, 1996.

[FG96b] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, January 1996.

[FG02] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer-Verlag, 2002.

[FLFMS03] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer-Verlag, 2003.

[FLMR03] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1):23–70, 2003.

[FT13] Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.

[GCT16] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 429–442, St Petersburg, FL, USA, January 2016. ACM Press.

[GM10] Georges Gonthier and Assia Mahbouhi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[GSM+11] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 227–240, Porto de Galinhas, Brazil, March 2011. ACM Press.

[Her12] Hugo Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *LICS*, pages 365–374. IEEE Computer Society, 2012.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [Lie04], pages 26–35.

[HS96] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[Hub16] Simon Huber. Canonicity for cubical type theory. *CoRR*, abs/1607.04156, 2016.

[HVC08] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In Doug Lea and Gianluigi Zavattaro, editors, *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION 2008)*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152, Oslo, Norway, June 2008. Springer-Verlag.

[Hyl91] J. M. E. Hyland. First steps in synthetic domain theory. In *Proceedings of the International Conference on Category Theory*, pages 131–156, Como, Italy, July 1991. Springer-Verlag.

[JJR06] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63:267–296, 2006.

[JLP+16] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The Definitional Side of the Forcing. In *LICS*, New York, United States, May 2016.

[JTS12] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. Extending type theory with forcing. In *Logic in Computer Science (LICS), 2012*, pages 395–404. IEEE, 2012.

[KF10] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, January 2010.

[KG13] Hsiang-Shang Ko and Jeremy Gibbons. Relational algebraic ornaments. In *Proceedings of the ACM SIGPLAN Workshop on Dependently Typed Programming (DTP 2013)*, pages 37–48. ACM Press, 2013.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[KIL+96] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. *arXiv preprint arXiv:1211.2851*, 2012.

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*, pages 49–58, St. Louis, MO, USA, 2005. ACM Press.

[Kri94] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994.

[Kri09] Jean-Louis Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

[Kri11] J.L. Krivine. Realizability algebras: a program to well order $\mathbb{R}$. *Logical Methods in Computer Science*, 7(3):1–47, 2011.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41(1):42–54, 2006.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant* Coq. PhD thesis, Université Paris-Sud, July 2004.

[Lev01] Paul Blain Levy. *Call-by-push-value*. PhD thesis, Queen Mary, University of London, 2001.

[Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.

[LS13] D. R. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 223–232, June 2013.

[LTD12] Paul Leger, Éric Tanter, and Rémi Douence. Modular and flexible causality control on the web. *Science of Computer Programming*, December 2012. Available online.

[Luo89] Zhaohui Luo. ECC, an extended calculus of constructions. In *LICS*, pages 386–395. IEEE Computer Society, 1989.

[Lur09] Jacob Lurie. *Higher topos theory*. Annals of mathematics studies. Princeton University Press, Princeton, N.J., Oxford, 2009.

[MBHJJ11] A. Mdhaffar, R. Ben Halima, E. Juhnke, and M. Jmaiel. AOP4CSM: An aspect-oriented programming approach for cloud service monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology (CIT)*, pages 363–370, 2011.

[McB10] Conor McBride. Ornamental algebras, algebraic ornaments. Technical report, University of Strathclyde, 2010.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 3–10, Nice, France, January 2007. ACM Press.

[Miq11a] A. Miquel. Forcing as a program transformation. In *Proceedings of the 26th Symposium on Logic in Computer Science (LICS'11)*, pages 197–206. IEEE, 2011.

[Miq11b] Alexandre Miquel. Forcing as a program transformation. In *LICS*, pages 197–206. IEEE Computer Society, 2011.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[MM92] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.

[MM12] Louis Mandel and Luc Maranget. *The JoCaml language Release 4.00*. INRIA, august 2012.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[MS08] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer-Verlag, 2008.

[NCT04] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [Lie04], pages 7–15.

[OSC10] Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [AOS10], pages 109–120.

[OSZ12] Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *Proceedings of the 6th workshop on Programming Languages Meets Program Verification (PLPV 2012)*, pages 3–14. ACM Press, 2012.

[OT13] Marko Obrovac and Cédric Tedeschi. Experimental evaluation of a hierarchical chemical computing platform. *International Journal of Networking and Computing*, 3(1):37–54, 2013.

[OTMW04] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

[PPT11] Jean-Louis Pazat, Thierry Priol, and Cédric Tedeschi. Towards a chemistry-inspired middleware to program the internet of services. *ERCIM News*, 85(34), 2011.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC: an aspect-oriented distributed dynamic framework. *Software—Practice and Experience*, 34(12):1119–1148, 2004.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL'98*, pages 378–390. ACM Press, 1998.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169. ACM Press, June 2008.

[SA16] Wouter Swierstra and Joao Alpuim. From proposition to program - embedding the refinement calculus in Coq. In *Proceedings of the 13th International Symposium on Functional and Logic Programming (FLOPS 2016)*, pages 29–44, Kochi, Japan, March 2016.

[Sch15] Gabriel Scherer. Multi-focusing on extensional rewriting with sums. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 317–331. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[SGR+10] Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology*, 20(2), August 2010. Article 5.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press. ACM SIGPLAN Notices, 37(11).

[SNI15] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 195–207, Mumbai, India, January 2015. ACM Press.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, August 2008.

[Soz07] Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07*, pages 13–24, Freiburg, Germany, 2007. ACM Press.

[SPAK10] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):Article 1, June 2010.

[ST06] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 481–497, Portland, Oregon, USA, October 2006. ACM Press. ACM SIGPLAN Notices, 41(10).

[Tab10] Nicolas Tabareau. A theory of distributed aspects. In AOSD 2010 [AOS10], pages 133–144.

[TFD+10] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Scoping strategies for distributed aspects. *Science of Computer Programming*, 75(12):1235–1261, December 2010.

[TFT13] Nicolas Tabareau, Ismael Figueroa, and Éric Tanter. A typed monadic embedding of aspects. In Jörg Kinzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 171–184, Fukuoka, Japan, March 2013. ACM Press.

[Tie72] Myles Tierney. Sheaf theory and the continuum hypothesis. *LNM 274*, pages 13–42, 1972.

[TJ06] Eddy Truyen and Wouter Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:147–181, 2006.

[Tri99] Todd Trimble. What are 'fundamental n-groupoids'? seminar at DPMMS, Cambridge, August 1999.

[TT06] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, June 2006. Springer-Verlag.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 60–76, Austin, TX, USA, January 1989. ACM Press.

[WDR14] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In José Pedro Magalhães and Tiark Rompf, editors, *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP 2014)*, pages 15–24, Gothenburg, Sweden, August 2014. ACM Press.

[Wer97] Benjamin Werner. Sets in types, types in sets. In *Theoretical aspects of computer software*, pages 530–546. Springer, 1997.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN Conference on Functional Programming (ICFP 2003)*, pages 127–139, Uppsala, Sweden, September 2003. ACM Press.