Université de Montréal

Typer a de la classe

Le polymorphisme ad hoc dans un langage avec des types dépendants et de la métaprogrammation

par

Jean-Alexandre Barszcz

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

10 mai 2021

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé

Typer a de la classe

Le polymorphisme *ad hoc* dans un langage avec
des types dépendants et de la métaprogrammation

présenté par

Jean-Alexandre Barszcz

a été évalué par un jury composé des personnes suivantes :

Marc Feeley
(président-rapporteur)

Stefan Monnier
(directeur de recherche)

Abdelhakim Hafid
(membre du jury)

Résumé

La modularité est un enjeu important en programmation, surtout quand on l'enrichit avec des preuves, comme dans les langages avec des types dépendants. Typer est un tel langage, et afin d'augmenter sa modularité et de lui ajouter un moyen de faire la surcharge d'opérateurs, on s'inspire d'Agda et Coq et on l'étend avec les arguments instances, qui généralisent les classes de types de Haskell. Un aspect qui distingue notre conception est que comme Typer généralise les définitions, la généralisation des contraintes de classe est grandement facilitée. Pour pouvoir faire les preuves de lois de classes, on doit également ajouter l'élimination dépendante des types inductifs au langage, dont certains aspects sont en retour facilités par les arguments instances. Sur la base de ces deux fonctionnalités, on offre également une solution au problème de la cécité booléenne, tel que décrit par Harper.

Mots clés : polymorphisme *ad hoc*, classes de types, types dépendants, métaprogrammation

Abstract

Modularity is an important concern for software development, especially when the latter is enriched with proofs in a language with dependent types. Typer is such a language, and in order to increase its modularity, and also provide a way to overload operators, we take inspiration from Agda and Coq and extend it with instance arguments, a generalization of Haskell's type classes. An aspect that sets our design apart is that since Typer generalizes definitions, it greatly simplifies the generalization of class constraints. In order to allow writing proofs for class laws, we must also implement the dependent elimination of inductive types. In return, instance arguments facilitate some details of dependent elimination. Using both features, we suggest a solution to the problem of Boolean Blindness.

Keywords : *ad hoc* polymorphism, type classes, dependent types, metaprogramming

Table des matières

Résumé 5

Abstract 7

Liste des sigles et des abréviations 13

Chapitre 0. Introduction 15

0.1. Contexte général 15

0.2. Problématique 17

0.2.1. Améliorations à Typer 17

0.2.2. Contributions à la programmation en général 19

0.3. Plan 22

Chapitre 1. Survol de Typer 23

1.1. Concepts de base 23

1.2. *S-exps* et Macros 24

1.3. Types dépendants 28

1.3.1. Intuition pour les types dépendants 28

1.3.2. Correspondance de Curry-Howard 29

1.3.3. Type d'égalité et effaçabilité 30

1.3.4. Des Sexps aux Lexps 31

1.4. Paramètres implicites et inférence 32

1.4.1. Bases de l'inférence 32

1.4.2. Typage bidirectionnel 33

1.4.3. Unification 33

1.4.4. Polymorphisme des `let` et généralisation 33

1.5. Structure de l'interpréteur Typer 34

Chapitre 2. Mécanismes de classes de types 35

2.1. Haskell : Traduction du dictionnaire et résolution ... 35
2.1.1. Résolution d'instances ... 36
2.1.2. Récursion de la résolution ... 37
2.1.3. Extensions aux classes de types en Haskell ... 37
2.2. Scala : Paramètres Implicites ... 40
2.2.1. Fonctionnement des paramètres implicites ... 40
2.2.2. Une alternative aux classes de types ... 41
2.2.3. Utilisation répandue ... 43
2.2.4. Implicites en Ocaml ... 44
2.3. Propriétés de la résolution ... 44
2.3.1. Haskell : cohérence et unicité globale ... 44
2.3.2. Retours en arrière et terminaison de la résolution ... 45
2.4. Types dépendants et lois de classes ... 46
2.5. Coq ... 47
2.5.1. Les classes de types ... 47
2.5.2. Les structures canoniques ... 50
2.6. Agda ... 50
Chapitre 3. Élimination des types inductifs ... 53
3.1. Élimination dépendante ... 53
3.2. Types inductifs à la Coq ... 54
3.3. Élimination dépendante en Coq ... 56
3.4. Patron du convoi ... 57
3.5. Élimination des types inductifs en Agda ... 58
3.6. Le patron *keep* ou *inspect*, une variation du convoi ... 60
3.7. Types inductifs dans Typer ... 61
Chapitre 4. Extension de Typer avec les classes de types ... 63
4.1. Choix de conception ... 63
4.1.1. Dictionnaires ... 63
4.1.2. Résolution locale ... 63

4.1.3. Inférence des contraintes de classe . . . 64
4.1.4. Classes *a posteriori* pour les preuves . . . 64
4.1.5. Résolution sans retours . . . 65
4.2. Détails de l'implantation . . . 65
4.2.1. Ensemble de classes de types . . . 66
4.2.2. Résolution d'instances . . . 66
4.2.3. Report de la résolution . . . 68
4.2.4. Contrôle des instances . . . 69
4.3. Extension de la bibliothèque de base . . . 69
4.3.1. Littéraux polymorphes . . . 69
4.3.2. Monades . . . 70
Chapitre 5. Amélioration de l'élimination des types inductifs en Typer . . . 73
5.1. Ajout des égalités dans les branchements . . . 74
5.1.1. Sémantique statique . . . 74
5.1.2. Sémantique dynamique . . . 74
5.1.3. Correction de l'exemple de la fonction `head` . . . 75
5.2. Macro pour l'élimination dépendante . . . 76
Chapitre 6. `Decidable`, une solution à la cécité booléenne . . . 79
6.1. Booléens en Typer . . . 79
6.2. Égalités comme solution à la cécité booléenne . . . 79
6.3. Distinction entre booléens et propositions . . . 81
6.4. Classe `Decidable` . . . 82
6.5. Un `If` pour `Decidable` . . . 83
6.5.1. Démonstration : la fonction `absurd` . . . 84
6.5.2. Le choix des ifs . . . 85
Chapitre 7. Évaluation . . . 87
7.1. Évaluation qualitative . . . 87
7.1.1. Extensions de Haskell . . . 87
7.1.2. Propriétés de la résolution en Typer . . . 88
7.1.3. Lois de classes . . . 90

7.1.4. Implicites généralisés 91
7.1.5. Utilisabilité de la solution proposée 92
7.2. Tests de performance 93
7.2.1. Cadre expérimental 93
7.2.2. Vérification empirique des complexités algorithmiques 94
7.2.3. Ralentissement raisonnable avec arguments instances 96
7.2.4. Aucun ralentissement sans arguments instances 97
7.3. Extension de la bibliothèque de base 98
Conclusion 101
Références bibliographiques 103

Liste des sigles et des abréviations

GADT : Type Algébrique Généralisé (Generalized Algebraic Data Type). 61

ITP : Assistant à la Preuve Interactif (Interactive Theorem Provers). 29, 47

MPTC : Classe à Plusieurs Paramètres (Multi-Parameter Type Class). 38, 39, 42, 87, 88

OOP : Programmation Orientée-Objet (Object-Oriented Programming). 16, 35

OPG : Grammaire de Précédence d'Opérateurs (Operator Precedence Grammar). 25

PTS : Système de Types Pur (Pure Type System). 28, 47

Chapitre 0

Introduction

0.1. Contexte général

C'est une idée bien établie dans le monde du génie logiciel qu'il est plus économe de corriger les défauts plus tôt dans le développement, dans leur phase d'apparition, que de laisser les dettes techniques accumuler de l'intérêt [**38, 63, 34, 43**].

Une façon de s'assurer que les bogues sont découverts aussi tôt que possible est d'ajouter aux compilateurs la responsabilité de vérifier que les programmes compilés respectent leur spécification. C'est là un des objectifs des langages fonctionnels statiquement typés. Dans ces langages, chaque fonction se fait associer un type, qui correspond essentiellement à une spécification, souvent approximative, du comportement désiré, et les types sont vérifiés durant la compilation, ce qui fait que beaucoup de défauts logiciels peuvent être signalés au programmeur au moment de l'écriture d'un programme. Un dicton populaire illustre cette idée : "Une fois que ça compile, généralement, ça fonctionne !". Des sondages montrent que la communauté du langage statiquement typé Haskell semble être généralement en accord avec cet adage [**18**]. Une étude de projets sur la forge logicielle Github confirme que le choix de langages fonctionnels statiquement typés réduit le nombre de *commits* corrigeants des bogues, bien que l'effet soit limité [**59**].

Par contre, les types peuvent parfois rendre le code plus verbeux. En pratique, on essaie souvent de les inférer à partir du reste du code. De cette façon, en Haskell, la définition de la fonction identité peut s'écrire `id x = x`, et le compilateur lui infère alors le type `a → a` pour un type `a` abstrait, puisque le type de `x` n'influence en rien la définition de `id`. On dit que le type de `id` est *généralisé* pour décrire cette quantification implicite (`id` fonctionne *pour tout* type `a`).

Les systèmes de types avec types dépendants permettent d'étendre l'expressivité des types et ainsi de renforcer les garanties offertes par le compilateur. Dans certains langagues ayant cette fonctionnalité, on peut faire des preuves formelles de l'exactitude des programmes écrits. Ces langages font alors partie de systèmes qu'on appelle des assistants à la preuve.

Cependant, le code qui contient à la fois des programmes qu'on veut exécuter, leur spécification, et des preuves exacerbe le problème de verbosité. Par exemple, seulement 13% de l'effort initial pour un compilateur certifié d'un sous-ensemble de du langage C consistait, en termes de lignes de code, de définitions de fonctions utiles à l'exécution, les 87% restants étant constitués de spécifications, de lemmes et de preuves [**37**]. Cela démontre l'importance des coûts de l'assurance qualité, et la nécessité de les réduire pour répandre la pratique.

Afin de faciliter leur développement, surtout à grande échelle, les projets de vérification utilisent des techniques similaires à celles qui sont répandues en génie logiciel pour rendre le code modulaire et favoriser sa réutilisation. En Programmation Orientée-Objet (OOP, Object-Oriented Programming), les développeurs suivent le principe d'encapsulation, c'est-à-dire d'isoler l'implantation de l'interface d'une composante logicielle [**57**]. En programmation fonctionnelle et dans les assistants à la preuve, les classes de types et les modules sont deux fonctionnalités de langages de programmation qui rendent possible la modularité et la réutilisation de code, et par extension, de preuves [**61**]. Le présent travail se concentre sur les classes de types, et leur relation aux modules sera discutée au chapitre 2.

Les classes de types ont d'abord été introduites dans le cadre du langage Haskell pour faire la surcharge d'opérateurs et de fonctions [**72**]. L'exemple classique est celui de la fonction polymorphe `==`, qui sert à vérifier si deux valeurs sont égales. Comme la notion d'égalité peut varier selon le type, et comme de nouveaux types peuvent toujours être définis, il serait peu pratique de définir `==` une fois pour toute. Les classes de types donnent un moyen d'étendre la définition de `==` à de nouveaux types. Ainsi, en Haskell, on peut définir la classe `Eq` des types qui supportent la comparaison avec `==` :

```
1 class Eq a where
2   (==) :: a → a → Bool
```

Cet extrait de code définit la *classe* `Eq`, qui, pour un type abstrait `a`, contient une fonction membre `==`, testant l'égalité de deux valeurs dans `a`. Le type de `==` est `Eq a ⇒ a → a → Bool`, qu'on peut comprendre de la façon suivante : si le type `a` a une instance de la classe `Eq`, alors `==` aura le type `a → a → Bool`. La partie du type qui précède la double flèche liste les *contraintes de classe*.

On peut alors définir `==` pour un nouveau type en donnant l'*instance* de `Eq` pour ce type. Par exemple, l'égalité des booléen peut être définie comme suit :

```
1   instance Eq Bool where
2     True  == True  = True
3     False == False = True
4     _     == _     = False
```

Alors qu'on a la définition de la classe et de l'instance, on peut utiliser == sur des valeurs booléennes, ainsi l'expression `True == False` donne `False`. Pour un exemple simple comme celui-ci, on aurait pu définir une fonction monomorphe pour les booléens comme `Bool_== :: Bool → Bool → Bool`, mais l'utilité des classes de types rayonne quand on les emploie de façon polymorphe. Par exemple, on peut définir une fonction de tri qui fonctionne avec n'importe quel type pour lequel on a une relation d'ordre : `sort :: Ord a ⇒ [a] → [a]`. C'est un polymorphisme qu'on appelle *ad hoc*, parce qu'il permet différents comportements selon le type, par opposition au polymorphisme dit paramétrique, qui fonctionne uniformément pour tous les types (comme dans le cas de `id`).

Dans un contexte où on a des types dépendants, il faut noter que les membres des classes de types peuvent être des fonctions, mais aussi des preuves de propriétés de ces fonctions. Pour notre exemple avec `Eq`, on pourrait imaginer que la classe contienne ==, ainsi qu'une preuve que == décrit une relation d'équivalence. Cette preuve devrait être fournie pour chaque définition d'instance, et pourrait être utilisée par le code qui agit polymorphiquement sur des `Eq`.

0.2. Problématique

On a mentionné que les classes de types ont une raison d'être supplémentaire avec les types dépendants puisqu'elles peuvent alors avoir des preuves comme membres. On verra au chapitre 2 qu'elles peuvent être basées sur un mécanisme qui permet également une forme de recherche automatique de preuves, ce qui redouble leur utilité dans ce contexte.

On verra également que les classes de types ont une interaction intéressante avec la fonctionnalité de généralisation, mentionnée dans le contexte de Haskell. Le langage Typer est un des rares systèmes qui offre la généralisation et les types dépendants, ce qui en fait donc un terrain attrayant pour expérimenter avec les classes de types. De plus, il a une syntaxe extensible et permet également de faire de la métaprogrammation, ce qui lui donne le potentiel de faciliter un style de programmation qui intègre la vérification tout en demeurant pratique. Un survol plus approfondi du langage et de ses concepts apparaît au chapitre 1. Notons d'abord quelques éléments de Typer qui pourraient être améliorés pour rendre le langage plus utile en pratique. Remarquons ensuite quelques difficultés dans la programmation en général qui motivent notre conception.

0.2.1. Améliorations à Typer

(1) Surcharge et réutilisation de code
Il manque un moyen de faire la surcharge d'opérateurs en Typer. Pour l'addition, par exemple, les programmeurs sont forcés d'utiliser des fonctions monomorphes comme

`Float_+`, qui, en comparaison avec une fonction `+` polymorphe, alourdissent l'utilisation du point de vue de la syntaxe et découragent l'écriture de code réutilisable. Similairement, les littéraux d'entiers sont monomorphes et pareil pour la notation `do` pour les monades, qui utilise directement la monade `IO`.

On veut donc augmenter la modularité, la réutilisabilité et le polymorphisme du code Typer, et permettre la surcharge en y ajoutant les classes de types.

(2) Variables sans nom, à bas prix

On a déjà mentionné le besoin d'omettre des types par souci de concision, et plutôt de laisser au compilateur de les inférer. Les types dépendants rendent ce problème plus important puisqu'ils peuvent ajouter beaucoup de code qui ne sert pas à l'exécution, mais plutôt à la vérification et aux outils de conception. On voudrait pouvoir écrire du code dans un style plus implicite, où les preuves sont omises et déduites par le compilateur. Ainsi, en Typer, on peut déjà définir des variables (qui peuvent correspondre à des preuves) implicitement, sans leur donner de nom. Par contre, le seul moyen actuel d'y accéder est de s'y référer par leur indice de de Bruijn (essentiellement l'adresse de la variable dans le contexte), ce qui est fragile et peu ergonomique.

On cherche donc à offrir un meilleur mécanisme pour faire référence à des preuves anonymes du contexte de façon implicite. Il sera expliqué au chapitre 4 que l'implantation des classes de types repose sur un mécanisme qu'on appelle les *arguments instances*, qui permet de se référer à des variables du contexte par leur type. Il s'agit donc directement d'un moyen qui facilite l'usage implicite de preuves, et les référence à des variables anonymes sans utiliser d'indices de de Bruijn.

(3) Égalités dans les branchements

Étant un langage supportant les types dépendants, Typer peut être utilisé pour favoriser la correctitude de programmes et pour y représenter des propositions intéressantes. Un exemple simple serait une variation de la fonction `hd` en Haskell, qui donne le premier élément d'une liste et lance une erreur quand la liste fournie est vide. Au lieu de lancer une erreur, on pourrait vouloir vérifier statiquement que la liste passée en paramètre ne peut pas être vide. On le fait en ajoutant un nouveau paramètre qui correspont à une preuve de cette propriété.

```
1  head : (ls : List ?τ) → (p : Not (Eq nil ls)) → ?τ ;
2  head ls p =
3    case ls
4    | cons x xs ⇒ x
5    | nil ⇒ ? ; %% Contradiction ! On sait que ls n'est pas nil !
```

Dans cet extrait de code, on retrouve une définition partielle d'une fonction `head`, qui prend deux paramètres : `ls` est une liste polymorphe d'éléments d'un type `?τ` fourni implicitement, et `p` est une preuve que l'argument précédent, `ls`, n'est pas `nil`, la

liste vide. L'implantation de la fonction procède par filtrage comme on pourrait le faire en Haskell avec une expression `case`. Dans le cas où la liste correspond à une paire (un *cons*), on peut simplement retourner le premier élément. Cependant, dans le cas ou la liste est vide, on a une contradiction, puisqu'on a une preuve qu'elle ne l'est pas !

Comment fait-on pour exprimer cette impossibilité et quel terme peut-on mettre à la place du point d'interrogation dans l'exemple ? On verra au chapitre 5 que ce problème est résolu par l'ajout de preuves d'égalité anonymes dans les branchements, et l'accès à ces preuves est facilité par l'implantation des arguments instances.

(4) Élimination dépendante

Dans les langages avec des types dépendants, il est essentiel de pouvoir raisonner par cas : si on veut prouver une proposition `P b` qui dépend d'une valeur booléenne `b`, il suffit de prouver séparément `P true` et `P false`. C'est une propriété qu'on appelle *élimination dépendante* puisqu'elle sert à éliminer des types inductifs en présence de dépendances. Par exemple, on peut commencer à prouver $P(b) = (b \vee \neg b)$ en essayant un filtrage sur `b` :

```
1  P b = Eq (or b (not b)) true;
2
3  preuve : (b : Bool) -> P b;
4  preuve b =
5    case b
6    | true => <Ptrue>
7    | false => <Pfalse>;
```

Intuitivement, on sait que la proposition devrait être satisfaite du côté gauche du `or` quand `b` est vrai, et du côté droit quand `b` est faux. Cependant, Typer n'offre pas de moyen d'exploiter la valeur de `b` connue dans chaque branche. En effet, les expressions `case` en Typer imposent le même type à chaque branche, ce qui l'empêche de dépendre de `b`. On n'arrive donc pas à raisonner par cas.

Afin de pouvoir exprimer l'élimination dépendante dans ce langage, on développe une solution qui exploite la métaprogrammation et les égalités dans les branchements, permettant le raisonnement par cas.

0.2.2. Contributions à la programmation en général

On cherche à améliorer la programmation en général en facilitant l'utilisation des types dépendants. Ceux-cis promettent de réduire les coûts de développement en attrapant plus de bogues au moment de la conception et en offrant plus d'informations aux outils de développement. Cependant, certains obstacles rendent leur adoption plus difficile. On s'attaque à trois d'entre eux :

(1) Classes de types avec généralisation et types dépendants

On a vu que Haskell généralise la définition de la fonction `id`. Cela évite le besoin d'écrire explicitement dans le code de la fonction qu'elle fonctionne pour n'importe quel type. Haskell généralise également les contraintes de classe : le compilateur peut déduire qu'une fonction accepte n'importe quel type d'une classe donnée et ajoute la contrainte de classe au type inféré pour la fonction. Par exemple, prenons la fonction de tri par insertion :

```
1 sort [] = []
2 sort (x :xs) = insert x (sort xs) where
3   insert x [] = [x]
4   insert x (y :ys) = if x < y then x :y :ys else y :insert x ys
```

Le type des éléments de la liste n'est pas intéressant au fonctionnement de `sort`, sauf pour le fait qu'on doit pouvoir comparer des éléments entre eux avec l'opérateur `<`. Cet opérateur fait partie de la classe `Ord`, et le type de `sort` est donc généralisé pour obtenir `sort :: Ord a ⇒ [a] → [a]`.

Typer est un des rares langages avec types dépendants qui offre la généralisation et ne demande pas de fournir explicitement de type de chaque définition. On étend cette généralisation aux classes de types, ce qui est utile pour la concision et permet également plus de flexibilité dans la maintenance du code, comme le démontrera l'exemple des *configurations implicites* au chapitre 7.

(2) Complexité de l'élimination des types inductifs

Dans la programmation statiquement typée, il est indispensable de permettre aux utilisateurs d'un langage de définir de nouveaux types. Classiquement, avec les types dépendants, on peut définir ce qu'on appelle des *types inductifs*. Ceux-cis permettent de décrire des structures de données, possiblement augmentées d'informations qui servent à vérifier des invariants. Cependant, certains raisonnements sur ces types inductifs sont difficiles à exprimer, ce qui a mené à l'émergence de patrons de conception comme le patron du convoi pour les contourner.

Le chapitre 3 expliquera que Typer offre les types inductifs, mais sous une forme un peu différente que celle qu'on retrouve classiquement dans les langages similaires. On complète cette conception d'une façon qui évite le patron du convoi, et qui simplifie donc certaines utilisations des types inductifs.

(3) Cécité booléene

Le concept de cécité booléenne (traduction de *boolean blindness*) a été popularisé par Harper, dans un article en ligne [**22**], puis dans son manuel [**23**]. Ce concept désigne le problème lié à l'utilisation des booléens qui est dû au fait qu'une valeur booléenne n'a pas intrinsèquement de signification. En effet, une telle valeur n'est utile que quand

on en connait la provenance. Considérons un exemple de code Haskell en commençant par rappeller quelques définitions :

```
1  data Maybe a = Just a | Nothing
2
3  isJust :: Maybe a → Bool
4  isJust (Just _) = True
5  isJust Nothing  = False
6
7  fromJust :: Maybe a → a
8  fromJust (Just v) = v
9  fromJust _ = undefined -- On soulève une erreur
```

Une valeur de type `Maybe a` peut contenir un `a` (le cas du constructeur `Just`) ou non (`Nothing`). La fonction `isJust` sert à vérifier si une valeur du type `Maybe a` contient effectivement un `a`, et dans ce cas, la fonction `fromJust` sert à l'extraire. Autrement, `fromJust` lance une erreur. On peut définir une version naïve de la fonction `maybe`, qui applique une fonction à un `Just a`, ou qui retourne une valeur par défaut, en utilisant les définitions précédentes :

```
1  maybe :: b → (a → b) → Maybe a → b
2  maybe d f m = if (isJust m) then f (fromJust m) else d
```

La personne qui écrit la définition de `maybe` ci-haut sait que l'appel à `fromJust m` est sans danger dans la branche `then`, puisque le booléen sur lequel le branchement s'effectue vient de l'expression `isJust m` et il doit être vrai. Alors, bien que la valeur `true` n'a pas intrinsèquement de signification, si on sait qu'elle est le résultat de l'appel `isJust m`, on apprend quelque chose au sujet de `m`. Cependant, puisque le système de types ne tient pas compte de la provenance du booléen, on peut accidentellement échanger les branches sans erreur de compilation. Donc, rien n'empêche le programmeur de se tromper et d'appeller `fromJust` dans la mauvaise branche.

D'autre part, Harper rappelle aussi la distinction parfois oubliée entre les valeurs booléenes et les propositions mathématiques. Cette distinction est importante dans un contexte constructif et où l'on ne néglige pas la calculabilité. Dans la bibliothèque de base du langage Coq, par exemple, on retrouve les deux versions des comparaisons d'entiers naturels : $\leq ?$ correspond au prédicat retournant un booléen, et $\leq$ correspond à une proposition.

On s'attaque à ces deux problèmes d'un coup, en montrant d'abord comment les types d'égalités peuvent permettre certains raisonnements sur la provenance de valeurs, puis en s'appuyant sur la classe `Decidable` pour remplacer beaucoup d'usages

de booléens et éviter la duplication d'opérateurs. En plus de fournir un exemple intéressant d'utilisation des arguments instances, notre solution à la cécité booléenne suggère une façon de programmer qui paraît plus idiomatique des types dépendants.

0.3. Plan

On commence par introduire Typer et ses concepts au chapitre 1. Ensuite, on dresse l'état de l'art au sujet des classes de types au chapitre 2, et de l'élimination des types inductifs au chapitre 3. Ces fondations permettent l'extention de Typer avec les classes de types au chapitre 4 et l'implantation de l'élimination dépendante, utile pour les preuves, au chapitre 5. Ces deux extension se croisent dans la solution à la cécité booléenne qu'on présente au chapitre 6. Finalement, le chapitre 7 contient une évaluation de l'aspect central de ce travail : l'implantation des classes de types.

Chapitre 1

Survol de Typer

Typer se trouve au croisement intéressant de plusieurs lignées de systèmes. D'une part, le langage s'inspire à la base de ML [**44**], ce qui peut être vu dans ses concepts de base décrits à la section 1.1. D'ailleurs, son implantation est écrite en Ocaml [**60**]. Aussi, comme le suggère son nom en -er, Typer hérite des idées de Scheme [**1**] (originalement appelé Schemer). La section 1.2 discute de son traitement de la syntaxe abstraite et son extensibilité par les macros. De plus, le langage puise aussi une partie de son inspiration des assistants à la preuve comme Coq [**3**] et Agda [**46**] en offrant des moyens de mélanger du code et des preuves de façon transparente par les types dépendants. Cet aspect est approfondi à la section 1.3. Finalement, la section 1.4 décrit comment l'inférence fonctionne en Typer. On résume la structure de l'interpréteur à la section 1.5.

Au cours du présent chapitre, on introduit aussi la syntaxe de Typer, qui sera utile à la compréhension des exemples présentés dans le reste de ce mémoire.

1.1. Concepts de base

Avant tout, Typer est un langage fonctionnel statiquement typé. Comme en ML, on y écrit des programmes en définissant des types de données et des fonctions. Par exemple, on peut y définir le type des listes de la façon suivante :

```
1 type List (α : Type)
2   | nil
3   | cons α (List α);
```

Dans cet extrait, le type `List` est défini avec le paramètre `α`, qui doit être un type. La définition fournit aussi deux façons de construire une liste : soit avec le constructeur de liste vide `nil`, ou avec le constructeur d'élément de liste `cons`, qui prend en paramètre l'élément courant de type `α` et le reste de la liste (de type `List α`).

On peut aussi définir des fonctions. Par exemple, la fonction qui applique une fonction à chaque élément d'une liste peut s'écrire comme suit :

```
1 List_map : (α : Type) ⇒ (β : Type) ⇒ (α → β) → List α → List β ;
2 List_map f ls =
3   case ls
4   | nil ⇒ nil
5   | cons x xs ⇒ cons (f x) (List_map f xs) ;
```

La première ligne donne la signature de la fonction `List_map` en décrivant son type après le simple deux points (`:`). Comme en Haskell ou Ocaml, les fonctions sont currifiées : elles acceptent un premier argument et retournent une fonction qui accepte le second, et ainsi de suite, jusqu'au résultat. Dans l'exemple, les deux premiers arguments sont les types `α` et `β`, puisque notre fonction est polymorphe. On préfère généralement omettre le types quand on applique la fonction, puisque ceux-ci peuvent être déduits du type des autres arguments. `α` et `β` sont donc des arguments implicites, ce qui est exprimé avec les flèches implicites `⇒`. Aussi, on leur a donné des noms (précédant le `:`) en plus de leur type, pour pouvoir y faire référence dans le reste de la signature. Les deux arguments suivants sont explicites (flèche simple `→`) et ne sont pas nommés. Ils correspondent à la fonction à appliquer dans la liste, de type `α → β`, et la liste d'entrée, de type `List α`. Enfin, le type de retour de la fonction suit la dernière flèche.

La définition de la fonction elle-même commence par une abstraction des paramètres explicites en les nommant `f` et `ls` (ligne 2). Puis on retrouve le corps de la fonction qui est une expression `case` qui effectue un filtrage sur le paramètre `ls`. Dans le cas ou `ls` est `nil`, on retourne `nil` et dans l'alternative, on applique `f` à l'élément courant et on appelle récursivement `List_map` avant de construire l'élément de liste qui correspond au résultat.

1.2. *S-exps* et Macros

Jusqu'à maintenant, la syntaxe des exemples ressemble plutôt à celle de Haskell ou Ocaml que celle des Lisps. Cependant, l'analyse syntaxique du code Typer rend une première forme du code en S-expressions, soit des listes ou le premier élément correspond à un opérateur et le reste sont ses arguments. À la différence des Lisps, les parenthèses ne sont pas significatives et ne servent qu'à corriger la précédence des opérateurs.

Par exemple, on peut attribuer à l'opérateur `+` des précédences de chaque côté avec la déclaration :

```
1 define-operator "+" 111 130 ;
```

Les nombres donnés en arguments fixent la précedence de l'opérateur de chaque côté et doivent être choisis pour bien interagir avec le reste des opérateurs. L'utilisation infixe de `+` peut alors être vue comme du sucre syntaxique pour la forme préfixe équivalente :

```
exemple₀ = 1 + 2 + 3; %% ⟹ 6
% est équivalent à la définition traduite (désucrée) :
exemple₁ = (_+_ 1 (_+_ 2 3)); %% ⟹ 6
```

Cela nous permet de poursuivre l'exemple de la section précédente et d'appliquer partiellement l'addition :

```
exemple₂ = List_map (_+_ 1) (cons 1 (cons 2 (cons 3 nil)));
%% ⟹ (cons 2 (cons 3 (cons 4 nil)))
```

Les S-expressions, qu'on nomme "Sexps" peuvent être comprises par leur type Ocaml :

```
type sexp =
  | Node of sexp * sexp list
  | Symbol of symbol
  | String of location * string
  | Integer of location * integer
  | Float of location * float
  | Block of location * pretoken list * location
```

Une Sexp peut être un noeud composé comme `(_+_ 2 3)`, un symbole comme `+`, ou un litéral de chaine de caractères, de nombre flottant ou d'entier comme `1`. On peut ignorer les blocs, qui ne seront pas intéressants pour le présent travail.

L'analyse syntaxique est assez simple et se fait à l'aide de Grammaire de Précédence d'Opérateurs (OPG, Operator Precedence Grammar), ce qui rend possible non-seulement les opérateurs infixes, mais aussi les opérateurs mixfixes comme `let_in_` : si deux opérateurs se suivent et leur priorités correspondent, ils sont combinés. Par exemple, `let` ne s'associe pas à gauche, et sa priorité à droite est 3, soit la même que la priorité à gauche de `in`. On peut donc utiliser une syntaxe similaire à celle de Haskell, et l'analyse syntaxique fournit des s-expressions qui ressemblent à celles des Lisps :

```
let a = 1 + 2 in a + 3; %% ⟹ 6
% est équivalent à l'expression traduite (désucrée) :
let_in_ (_=_ a (_+_ 1 2)) (_+_ a 3); %% ⟹ 6
```

On peut noter l'existence du symbole vide, qu'on appelle *epsilon* et qu'on note `()` en Typer. Ce symbole est utilisé pour combler les "trous" quand il manque d'opérandes à un opérateur. Ainsi, l'expression `(+ 1)` est traduite à `(_+_ () 1)`, puisque `+` attend un argument de chaque côté, mais la parenthèse de gauche indique que l'argument de gauche est absent puisqu'elle appartient à une expression plus large, et `()` est fourni par défaut.

Finalement, Typer offre les macros sur les Sexps. Celles-ci sont traitées à l'étape suivant l'analyse syntaxique : l'élaboration. Quand Typer élabore une Sexp et que le symbole à la tête de celle-ci correspond à une définition ayant le type prédéfini `Macro`, l'expansion de celle-ci est faite. Le type `Macro` est simplement un type enveloppant une fonction de type `List Sexp → IO Sexp` :

```
1 type Macro
2   | macro (List Sexp → IO Sexp);
```

Pour faire l'expansion de la macro, la fonction associée à la variable de type `Macro` est évaluée avec la liste des Sexps constituant le reste du noeud de l'appel de macro. Un exemple utile de macro serait celle qui permet la syntaxe de section d'opérateur, c'est-à-dire qui rend possible l'omission d'opérandes en ajoutant des abstractions. L'objectif est de pouvoir écrire `(+ 1)` et que `+` corresponde à une macro qui traduit l'application en une lambda qui prend en paramètre les arguments manquants.

On peut commencer par définir une fonction auxiliaire qui traverse la liste des arguments et qui remplace les epsilons par des symboles créés à la volée avec la fonction `gensym`. Ces symboles sont enregistrés dans une autre liste. La fonction retourne son résultat dans la monade `IO`, puisqu'on en a besoin pour `gensym`. Le résultat est une paire de listes de Sexps : la première est la liste de symboles créés et la seconde est la liste des arguments modifiés.

```
1  section-fold-args : List Sexp → IO (Pair (List Sexp) (List Sexp));
2  section-fold-args args =
3    case args
4    | nil ⇒ IO_return (pair nil nil)
5    | cons arg args ⇒
6      do {
7        res ← section-fold-args args;
8        case Sexp_wrap arg
9        | symbol "" ⇒
10         do { sym ← gensym ()
11            ; IO_return (pair (cons sym (fst res)) (cons sym (snd res)))}
12       | _ ⇒ IO_return (pair (fst res) (cons arg (snd res)));
13       };
```

Cette fonction effectue d'abord un filtrage sur la forme de la liste d'arguments : s'il s'agit d'une liste vide, alors on retourne une paire de listes vides (ligne 4) ; autrement, on poursuit le traitement des arguments récursivement en traitant d'abord le reste de la liste (ligne 7). On constate en passant l'usage de la syntaxe `do`, qui permet de séquencer les actions dans une monade similairement à ce qui se fait en Haskell. Finalement, on filtre l'argument courant, et s'il s'agit du symbole sans nom (epsilon), on le remplace par un symbole généré par `gensym`, qu'on ajoute aussi à la liste des symboles générés (ligne 11). Autrement, on ajoute l'argument courant à la liste des arguments, sans modification (ligne 12).

On peut alors définir le corps de notre macro, soit la fonction du type `List Sexp → IO Sexp` qui prend la liste des arguments et bâtit la Sexp résultante :

```
section-make-op : String → List Sexp → IO Sexp;
section-make-op name args =
  do {
    p ← section-fold-args args;
    let vars : List Sexp;
        vars = fst p;
        body : Sexp;
        body = Sexp_node (Sexp_symbol name) ((snd p));
    in
        IO_return (Sexp_node (Sexp_symbol "lambda_→_") (List_concat vars (cons body nil)));
    };
```

Ici, on utilise la currification en définissant la fonction `section-make-op`, qui lorsque appliquée à la chaine correspondant à un opérateur, retourne une fonction du type attendu pour des macros. Cette fonction bâtit une abstraction de la liste des arguments retournée par `section-fold-args`. La S-expression correspondant à une abstraction à plusieurs arguments est un noeud dont le symbole de tête est `lambda_→_`, et les arguments sont les variables de l'abstraction, sauf le dernier, qui constitue le corps de la `lambda`.

Enfin, on définit `_+_` comme une macro et on fait référence à la fonction équivalente prédéfinie `##Int.+`, pour éviter la circularité. On peut ensuite utiliser des section de notre opérateur.

```
_+_ = macro (section-make-op "##Int.+");

exemple₃ = List_map (+ 1) (cons 1 (cons 2 (cons 3 nil)));
%% ⟹ (cons 2 (cons 3 (cons 4 nil)))
```

Dans l'exemple, quand l'élaboration rencontre la Sexp `(+ 1)`, traduite à `(_+_ () 1)`, et que `_+_` est définie comme une macro, la macro est appliquée à la liste des arguments (`()` et `1`), et retourne la nouvelle sexp `(lambda_→_ #gensym0 (##Int.+ #gensym0 1))` (où `#gensym0` serait le symbole unique retourné par `gensym`). On a donc obtenu la fonction qui implante la section `(+ 1)`.

On en profite pour noter que la syntaxe utilisée pour définir les types jusqu'à maintenant est aussi du sucre syntaxique implanté par une macro. Ainsi, la définition du type `List` donnée à la section 1.1 se traduit en plusieurs définitions :

- Une première pour le type lui-même :

```
List = typecons (List (α : Type)) (nil) (cons α (List α));
```

- Et ensuite une pour chaque constructeur :

```
1 nil  = datacons List nil;
2 cons = datacons List cons;
```

Ces définitions illustrent le caractère structurel plutôt que nominal du typage en Typer.

1.3. Types dépendants

On a déjà mentionné que Typer est un langage supportant les types dépendants, et que cela nous permet de faire des preuves de correctitude de programmes, similairement à ce qui se fait dans d'autres assistants à la preuve. Il est important de clarifier que Typer n'est pas encore doué de vérification de la totalité. Au niveau de la programmation, cela veut dire qu'on peut écrire des boucles infinies par erreur. Au niveau de la logique, cela signifie qu'on ne peut écarter le risque de raisonnement circulaire. Ainsi, on utilise le mot "preuve" avec liberté.

1.3.1. Intuition pour les types dépendants

Les types dépendants désignent le fait que les types peuvent dépendre de valeurs. Pratiquement, les calculs au coeur des langages de programmation fonctionnelle pure sont basés sur des lambda-calculs typés. Par exemple, ML et Haskell se basent sur le système de types Hindley-Milner, où l'on peut définir des valeurs dépendants d'autres valeurs (abstractions lambda) et, en respectant certaines conditions, on peut avoir des valeurs dépendants de types (fonctions polymorphes). Ainsi, le système Hindley-Milner se trouve entre le lambda-calcul simplement typé, qui ne permet pas les abstractions de types, et le Système F, qui les permet toujours. Cet axe est le premier de trois axes de dépendances possibles dans ce qu'on appelle le lambda-cube. Les deux autres axes du cube sont celui qui permet les opérateurs de types (types dépendants de types) et les types dépendants (types dépendants de valeurs). Au sommet du cube, on retrouve le Calcul des Constructions, qui permet les trois types de dépendances (en plus des valeurs dépendant de valeurs, présentes dans tous ces systèmes) [**57**].

Le lambda-cube classifie les calculs où on sépare syntaxiquement les catégories des types et des termes (ou valeurs). Typer est un Système de Types Pur (PTS, Pure Type System), ce qui veut dire que ces deux catégories sont combinées en une seule. Donc, les valeurs et types Typer sont tous des termes qui peuvent dépendre d'autres termes. Une hiérarchie est réintroduite, puisque le calcul garde une trace d'un niveau (essentiellement un nombre naturel) associé à chaque terme pour éviter les circularités et paradoxes.

Dans un article décrivant la conception de Typer [**45**], Monnier montre l'exemple classique des listes dont la longueur est connue statiquement :

```
type NList (a : Type) (n : Nat)
  | nil (p ::: Eq n zero)
  | cons a (NList a ?n) (p ::: Eq (succ ?n) n);
```

Chaque constructeur d'une telle liste porte une preuve d'égalité au sujet de sa longueur. Ainsi, le constructeur de liste vide `nil` porte une preuve que la longueur `n` est 0, et le constructeur d'élément de liste `cons` porte une preuve que la longueur est un de plus que celle du reste de la liste.

Plus généralement, le type des paramètres `p` (`p` pour preuve) de chacun des constructeurs est l'égalité entre `n`, la valeur d'un paramètre du type, et un autre entier naturel. Cette dépendance sur la valeur de `n` permet d'enregistrer, statiquement et au niveau des types, la longueur de la liste, et d'utiliser cette information pour éviter des erreurs de bornes, par exemple.

1.3.2. Correspondance de Curry-Howard

Une seconde intuition utile pour approcher les types dépendants est celle de la correspondance de Curry-Howard. Cette correspondance établit des liens étroits entre la programmation et la logique. C'est donc un cadre intéressant pour comprendre les preuves dans les Assistants à la Preuve Interactifs (ITPs, Interactive Theorem Provers). C'est aussi un concept nécessaire pour suivre l'application des classes de types à la classe `Decidable` pour répondre au problème de la cécité booléenne (chapitre 6).

La correspondance de Curry-Howard donnent une interprétation logique aux programmes et vice-versa. Les types correspondent à des formules de logique, et les programmes ayant ces types correspondent à des preuves de ces formules. Cela veut dire qu'un type non-habité correspond à une formule improuvable. Par exemple, il n'y a pas de programmes du type `Void` (également appellé `False`), et la formule logique correspondante, $\bot$ ou *Faux*, n'a pas de preuve.

Voici une façon d'encoder la correspondance formules/types :

(1) Comme mentionné, la formule $\bot = \textit{Faux}$ correspond au type vide `Void` qui n'est pas habité : on ne peut pas prouver *Faux*.

(2) Similairement, la formule $\top = \textit{Vrai}$ correspond au type unitaire `Unit` (parfois appellé `True`) qui est trivialement habité par le terme `()`.

(3) L'implication $A \to B$ correspond à une fonction `α → β` quand A correspond à `α` et B à `β`. L'application correspond alors au *modus ponens* :

```
app : (α : Type) ⇒ (β : Type) ⇒ (α → β) → α → β;
app f x = f x;
```

(4) La négation $\neg A$ d'une formule A peut être traduite à $A \to \bot$: on peut prouver qu'un type n'est pas habité en donnant une fonction qui mène de ce type à `Void`. C'est une forme de raisonnement par l'absurde.

(5) La quantification universelle, comme $\forall x.P(x)$, correspond aux flèches dépendantes `(x : ?t) -> ?P x`. L'application permet aussi de l'éliminer :

```
app_dep : (t : Type) => (P : t -> Type) => ((x : t) -> P x) -> (x : t) -> P x;
app_dep f x = f x;
```

(6) La conjonction logique $A \wedge B$ correspond aux paires : ayant une preuve de A et une preuve de B, on prouve $A \wedge B$. Les paires sont exprimées avec un type inductif en Typer :

```
type Pair (α : Type) (β : Type)
  | pair (fst : α) (snd : β);
```

(7) La quantification existentielle $\exists x.P(x)$ correspond au paires dépendantes :

```
type Exists (t : Type) (P : t -> Type)
  | exists (witness : t) (proof : P witness);
```

(8) La disjonction $A \vee B$ correspond au type somme, aussi un type inductif :

```
type Either (α : Type) (β : Type)
  | left α
  | right β;
```

1.3.3. Type d'égalité et effaçabilité

Le type dépendant le plus important dans le code Typer est le type prédéfini d'égalité `Eq`. `Eq x y` est habité si `x` et `y` sont la même valeur. La seule façon de construire une preuve d'égalité dans le calcul de base de Typer est avec la réflexion : pour tout terme `x`, `x` est égal à lui-même. La valeur prédéfinie `Eq_refl` correspond à cet axiome.

```
Eq_refl : (t : Type) ≡> (x : t) ≡> Eq x x;
```

C'est le premier exemple de la triple flèche. Cette flèche désigne son argument comme étant à la fois implicite (comme pour `⇒`) et effaçable. Les arguments effaçables ne sont là que pour la vérification de types et sont effacés avant l'évaluation. Ils ne peuvent donc pas être utilisés dans des contextes non-effacés, comme le sujet d'un branchement au moment de l'exécution.

On élimine les égalités avec la fonction prédéfinie `Eq_cast`, qui permet de transporter une formule d'un côté à l'autre de l'égalité :

```
Eq_cast : (x :?t) ≡> (y :?t) ≡> (p : Eq x y) ≡> (f :?t →?) ≡> f x → f y;
```

Cet axiome indique que si `x` est égal à `y`, pour tout opérateur `f`, on a que `f x` implique `f y`. On remarque que l'égalité passée en paramètre est effaçable.

Comme les égalités ne peuvent être construites qu'avec `Eq_refl` en Typer, elles n'ont pas de contenu en terme de calcul au moment de l'exécution. Elles ne servent qu'à la vérification de types et seront généralement en position effaçable dans le code présenté. D'ailleurs, on peut définir une fonction qui prend une égalité effacée et la réifie à l'exécution :

```
Eq_unerase =
  lambda x y (p : Eq x y) ≡>
  Eq_cast (p := p) (f := Eq x) Eq_refl; % Résultat : Un `Eq x y` non-effacé
```

Ici, on utilise l'élimination et l'introduction de l'égalité pour transporter la formule `Eq x` de `x` à `y`, comme `Eq_refl` peut ne pas être effacée, on peut obtenir une version non-effacée de l'égalité `p` fournie en paramètre. L'existence de cette fonction démontre que l'égalité n'a pas de contenu en terme de calcul au niveau de l'exécution, en Typer.

1.3.4. Des Sexps aux Lexps

Maintenant qu'on a vu plusieurs exemples d'expressions Typer, on peut clarifier la syntaxe abstraite du calcul au coeur du langage. Celle-ci est décrite par le type de données `lexp` (pour lambda-expression) en Ocaml. Les s-expressions sont traduites vers `lexp` par un processus qu'on appelle l'élaboration. On a déjà mentionné que l'élaboration effectue l'expansion des macros. Elle sert aussi à inférer tous les types, comme le décrit la prochaine section. Cela fait en sorte que toutes les variables dans `lexp` sont typées. Voici une définition simplifiée de `lexp` :

```
type ltype = lexp (* Les types sont des expressions comme des autres. *)
 and lexp =
   | Imm of sexp
   | Builtin of symbol * ltype
   | Var of vref
   | Let of def list * lexp
   | Arrow of param * ltype
   | Lambda of param * lexp
   | Call of lexp * (arg_kind * lexp) list
   | Inductive of label
                * (param list) (* Paramètres du type *)
                * (param list) map (* Les constructeurs et leurs paramètres *)
   | Cons of lexp * symbol
```

```
  | Case of lexp * ltype
            * ((arg_kind * vname) list * lexp) map
            * (vname * lexp) option
and param = arg_kind * vname * ltype
and def = vname * lexp * ltype
```

Les `lexp` comprennent :

- `Imm` : les valeurs de `sexp` immédiates pour les littéraux (comme des chaînes, entiers, etc.) ;
- `Builtin` : les constantes (de types, valeurs ou fonctions) prédéfinies comme `Eq`, `Eq_refl` ou `Eq_cast` ;
- `Var` : les variables ;
- `Let` : les listes de définitions mutuellement récursives avec une expression qui peut s'y référer, où chaque définition a un nom, une valeur et un type ;
- `Lambda` et `Arrow` : les fonctions et leur type flèche, où chaque paramètre a un `arg_kind` qui indique s'il s'agit d'un argument normal, implicite ou effacé, ainsi qu'un nom et un type ;
- `Call` : les appels de fonction qui sont composés de la fonction et de la liste d'arguments ;
- `Inductive` : les types définis par l'utilisateur qui ont un nom, une liste de paramètres pour le type, et une table qui associe les noms des constructeurs à leurs paramètres ;
- `Cons` : les valeurs de constructeurs, se référant à un type donné et le nom d'un de ses constructeurs ;
- `Case` : les expressions de filtrage, qui servent à déstructurer des types inductifs, et qui sont composées d'une expression à filtrer, d'un type de retour, d'une table de branches où chaque constructeur est associé à une liste de variables à lier ainsi qu'un corps de branche, et optionnellement d'une branche par défaut.

1.4. Paramètres implicites et inférence

Typer se distingue d'autres langages avec des types dépendants par sa gestion de l'inférence. De ce côté, en plus d'utiliser l'approche du typage bidirectionnel, il s'inpire beaucoup du système de type Hindley-Milner en implantant le polymorphisme des `let`, et en effectuant l'unification à la volée.

1.4.1. Bases de l'inférence

L'inférence sert à deviner les termes omis par le programmeur. Dans les langages basés sur Hindley-Milner, on omet généralement les types des variables, et ceux-ci sont inférés par le compilateur. Avec les types dépendants, l'inférence ne se limite pas qu'aux types. Plus généralement, n'importe quel terme Typer peut être omis en le remplaçant par une variable

d'unification, qu'on appelle aussi parfois "métavariable", et qu'on note avec un identifiant débutant par un point d'interrogation (`?`). Le système de type fait alors de son mieux pour trouver le terme qui devrait y être inséré, mais peut échouer, auquel cas, une erreur est signalée et le programmeur doit ajouter des annotations.

1.4.2. Typage bidirectionnel

Typer utilise l'approche bidirectionnelle [**58, 16**] pour élaborer ses termes et leur assigner des types en favorisant un flot d'information local. Cela veut dire que l'élaboration se fait par deux fonctions mutuellement récursives, l'une qui synthétise le type d'un terme (`infer`) et l'autre qui vérifie que le terme a un type donné (`check`). Toute l'information de typage passe soit par le type attendu (dans le cas `check`) ou retourné (dans le cas `infer`), soit par le contexte de typage. Le flot d'information est donc toujours local, ce qui est plus simple à comprendre pour l'usager et ce qui favorise la simplicité des messages d'erreur.

1.4.3. Unification

Quand on fait l'élaboration d'un appel, les arguments implicites qui sont omis sont remplacés par des variables d'unification. Par exemple, on peut appliquer la fonction d'identité `id : (t : Type) => t -> t` à un entier, ce qui requiert l'insertion d'une variable d'unification pour l'argument de type omis : `id (t := ?t) 1`. Au traitement de l'argument suivant, le type attendu `?t` du paramètre est unifié avec le type de l'argument `Int`. Puisque les deux doivent être identiques pour que l'appel soit bien typé, on assigne la valeur `Int` à la métavariable.

Pour résoudre les problèmes d'unification entre des paires de termes qui peuvent contenir des métavariables, on emploie un algorithme d'unification *ad hoc* assez simple, mais qui semble être utilisable en pratique.

1.4.4. Polymorphisme des `let` et généralisation

Pour définir la fonction d'identité polymorphe, on peut écrire toutes les abstractions de type explicitement en Typer :

```
1 id : (t : Type) => t -> t;
2 id = lambda t => lambda (x : t) -> x;
```

Ce style explicite paraît verbeux en comparaison avec la définition dans un système basé sur Hindley-Milner comme Haskell, où la même définition est abrégée à `id x = x`.

Avec Hindley-Milner, les quantification de types sont invisibles, on ne peut pas les noter explicitement. Pour pouvoir appliquer une fonction à des types différents, les définitions apparaissant dans des `let` sont généralisées. C'est à dire qu'implicitement, les variables

d'unification qui demeurent libres (comme le type inconnu de `x`), et qui n'apparaissent pas ailleurs, sont remplacées par des variables de type quantifiées universellement. C'est ce qu'on appelle le polymorphisme des `let` (ou *let-polymorphism* en anglais).

On procède de façon similaire en Typer, ce qui permet de raccourcir la définition à `id x = x`, et le bon type est inféré comme en Haskell. Alternativement, on peut aussi fournir une signature abrégée : `id : ?t → ?t`. Les signatures sont également généralisées, ce qui fait que comme la métavariable `?t` n'est contrainte d'aucune manière, et qu'elle échappe donc à l'unification, elle est automatiquement universellement quantifiée.

Finalement, on note que si la signature d'une définition a des paramètres implicites qui ne sont pas mentionnés par la définition, ceux-ci sont automatiquement ajoutés. Par exemple, dans le code suivant, on ajoute automatiquement un `lambda` implicite à la définition pour qu'elle corresponde à sa signature :

```
1 id : (t : Type) => t → t;
2 id x = x;
```

D'autres langages avec types dépendants, comme Agda, n'implantent pas la généralisation, et demandent plutôt qu'une signature soit fournie pour chaque définition.

1.5. Structure de l'interpréteur Typer

Pour résumer ce chapitre, on propose une vue d'ensemble de l'interpréteur Typer. D'abord, l'analyse syntaxique transforme le flot de lexèmes en s-expressions. Les s-expressions sont alors traitées par l'élaboration. L'élaboration est responsable de l'expansion des macros, de transformer les Sexps en Lexps en affectant un type à chaque sous-expression, et d'effectuer l'inférence pour deviner les types manquants. Ensuite, les types sont revérifiés dans le calcul coeur de Typer, entre autres pour se parer d'erreurs possibles dans l'élaboration. Après la vérification des types, on effectue l'effacement pour obtenir des termes dans un lambda-calcul non typé. Finalement, on peut évaluer ces derniers.

Chapitre 2

Mécanismes de classes de types

Ce chapitre tente de faire un portrait de l'état de l'art au sujet des classes de types, et servira de cadre pour l'extension de Typer avec cette fonctionnalité au chapitre 4. On commence par voir leur origine et un tour des concepts de base en Haskell, puis leur forme généralisée en Scala. Ensuite, on s'attarde aux propriétés qui permettent d'évaluer et de contraster différentes implantations. Finalement, on étudie l'influence des types dépendants et on se penche sur deux systèmes qui les utilisent : Coq et Agda.

2.1. Haskell : Traduction du dictionnaire et résolution

Les classes de types ont d'abord été introduites en Haskell pour faire la surcharge d'opérateurs, comme pour l'exemple abordé dans l'introduction avec l'opérateur d'égalité. Elles ont été présentées dans l'article fondateur de Wadler et Blott [**72**] avec des idées semblables à celles développées indépendamment par Kaes [**33**].

En plus d'introduire les concepts et la syntaxe des classes de types, cet article fondateur introduit la *traduction du dictionnaire*. Celle-ci permet d'exprimer le comportement d'un langage avec les classes de types à l'aide d'un langage plus simple qui n'offre pas cette fonctionnalité.

Cette traduction prend son nom du fait que les classes de types rassemblent plusieurs méthodes ou déclarations reliées dans une seule interface. Cet aspect leur vaut une ressemblance à d'autres fonctionnalités qui permettent l'aggrégation et le nommage, comme les dictionnaires, les enregistrements, les modules, ou les objets en OOP. À titre d'exemple, en Haskell, on peut traduire la classe `Eq` par un type de donnée `EqDict` qui constitue son "dictionnaire" :

```
1 data EqDict a = EqDict (a → a → Bool)
2
3 eq :: EqDict a → a → a → Bool
4 eq (EqDict f) = f
```

```
5
6  boolEqDict :: EqDict Bool
7  boolEqDict = EqDict boolEq where
8    boolEq True True = True
9    boolEq False False = True
10   boolEq _ _ = False
```

Le type `EqDict` encapsule la seule méthode de `Eq` dans un champ de la donnée. On déclare aussi l'accesseur `eq` qui obtient la fonction d'égalité à partir d'un dictionnaire. Remarquons que `eq` correspond à la traduction de `== :: Eq a => a -> a -> Bool`, puisque les contraintes de classe sont traduites par des dictionnaires passés en paramètre. Enfin, `boolEqDict` correspond à la traduction de l'instance de `Eq` pour les booléens.

2.1.1. Résolution d'instances

En plus d'aggréger des déclarations, un mécanisme de classes de types doit permettre la *résolution* d'instances. Par exemple, quand on utilise un opérateur surchargé comme dans l'expression `a == 0`, où `a` et `0` sont des `Int`, le système doit pouvoir déduire que l'opérateur est appliqué à des entiers, et donc trouver une instance `Eq Int` qui donne une signification à l'expression. La résolution décrit donc le choix des instances (ou dictionnaires) utilisées pour satisfaire les constraintes de classe.

Les décisions liées à la résolution doivent se faire durant la traduction, puisqu'il n'y a plus de classes de types dans son langage cible. Concrètement, puisque la traduction ajoute des paramètres aux fonctions qui ont des contraintes de classes, elle doit aussi ajouter les arguments correspondants dans leurs applications, soit les bons dictionnaires d'instance. Pour illustrer, afin de traduire `True == False`, on choisit l'instance `boolEqDict`, et l'expression devient `eq boolEqDict True False`.

Une question se pose quand on veut traduire l'opérateur `==` lui même, puisqu'on peut se demander s'il faut l'appliquer à une instance ou non. On se trouve devant deux résultats possibles : `eq :: EqDict a -> a -> a -> Bool` ou bien `eq boolEqDict :: Bool -> Bool -> Bool`.

Comment décider où appliquer les arguments qui correspondent aux instances durant la traduction, et comment les choisir ? Le principe qui guide ces décisions dans l'article de Wadler et Blott, ainsi que dans les spécifications du langage Haskell, est de choisir l'option qui donne le type le plus général, autrement dit, le type principal dans le système Hindley-Milner [**72, 25, 54**]. Ainsi, le type de l'opérateur d'égalité devrait être `Eq a => a -> a -> Bool` plutôt que `Bool -> Bool -> Bool`, et on choisit la traduction correspondante, même si une instance `Eq Bool` est présente dans le contexte et qu'elle pourrait être sélectionnée.

Le typage principal mène ces auteurs à quelques contraintes dans la conception. En effet, imaginons qu'une expression définisse localement la classe `Eq`, et retourne l'opérateur `==`, ce qui n'est pas permis en Haskell :

```
let class Eq a where ...
    instance Eq Bool where ...
  in (==)
```

Le type de l'expression ne pourrait pas mentionner `Eq`, qui ne serait plus dans la portée à l'extérieur de la définition locale ! Cela veut dire que le type principal ne serait pas permis. Afin de satisfaire cette condition de typage principal, Wadler et Blott imposent donc que les classes et instances soient définies globalement. Par ailleurs, afin qu'il n'y ait pas d'ambiguïté, leurs instances ne peuvent pas se chevaucher.

Le principe des types principaux fournit un objectif précis mais abstrait. Hall *et al.* formalisent la traduction des classes de types de Haskell vers un lambda-calcul polymorphe, détaillant ainsi la sémantique en considérant comment l'implantation peut se faire [**20**].

2.1.2. Récursion de la résolution

On veut parfois définir une instance pour un type polymorphe comme celui des listes. On peut vérifier l'égalité de deux liste en les parcourant élément par élément, à condition de pouvoir comparer les éléments entre eux ! L'instance elle-même impose donc une contrainte de classe : `instance Eq a ⇒ Eq [a] where ...`. Cette instance se traduit par une fonction `EqDict a → EqDict [a]`, puisque la contrainte devient un paramètre. De plus, quand on obtient cette instance par résolution, il faut également résoudre sa contrainte. La résolution se poursuit donc récursivement jusqu'à ce que toutes les contraintes soient satisfaites. Par exemple, quand on compare deux listes d'entiers avec l'expression `[1] == [2]`, on trouve d'abord l'instance pour les listes, ajoutant la contrainte `Eq Int`, qui trouve alors également son instance.

2.1.3. Extensions aux classes de types en Haskell

Le rapport spécifiant la première version de Haskell décrit comment la théorie présentée jusqu'ici s'applique dans le détail en Haskell [**25**]. Il étend les classes de types avec la possibilité de fournir des définitions par défaut pour les membres d'une classe, et avec un mécanisme pour dériver les instances de certaines classes prédéfinies de façon automatique.

Jusqu'à maintenant, la présentation des classes de types reste assez limitée : elles n'ont qu'un seul paramètre, qui doit avoir le *kind* `*` (autrement dit, il ne peut pas être un opérateur de type), et les membres des classes ne peuvent être que des valeurs. Depuis lors, de nombreuses extensions ont été ajoutées à l'implantation GHC de Haskell pour éviter ces limitations. Peyton Jones *et al.* ont exploré différentes extensions possibles et noté leurs avantages et inconvénients [**55**]. Un historique général de Haskell situe temporellement certains de ces développements dans leur contexte [**24**]. En voici quelques uns :

(1) Classes de constructeurs de types

Les classes de constructeurs de types (en anglais *constructor classes*) sont une évolution intéressante des classes de types dans un contexte où on permet le *higher-kinded polymorphism* [**30, 29**]. En bref, l'idée consiste à ne pas limiter les paramètres des classes à des types (de *kind* `*`), mais de les étendre à des constructeurs de type, autrement dit, des opérateurs sur les types, ayant un *kind* différent de `*`, tel que `*->*`. Par exemple, `List` (noté `[]` en Haskell) prend un paramètre de type et retourne un type quand on l'applique. Son *kind* est donc `*->*`. On peut vouloir "classer" l'opérateur `List` : les listes permettent, entre autre choses, d'appliquer une fonction à chaque élément. Une classe peut être définie pour la fonction `fmap`, qui permet d'appliquer une fonction à chaque élément d'une structure donnée :

```
1 class Functor f where
2     fmap :: (a -> b) -> f a -> f b
```

Le paramètre `f` de cette classe est appliqué à des types dans la signature de `fmap`. C'est donc un opérateur de type, et `Functor` est une classe de constructeurs de types. L'instance pour les listes est bien simple, elle fait appel à la fonction spécialisée pour les listes `map :: (a -> b) -> [a] -> [b]` :

```
1 instance Functor [] where
2     fmap = map
```

Une autre classe de constructeurs de types importante est celle des monades. `Monad` a d'ailleurs `Functor` comme superclasse, c'est-à-dire que chaque monade doit aussi avoir une instance de `Functor`. Étant donné l'importance centrale de `Monad` en Haskell (puisque les entrées et sorties d'un programme passent toutes par la monade `IO`), les classes de constructeurs sont aussi importantes et elles ont été intégrées dans Haskell 1.3 dès 1996 [**24**].

(2) Classes avec plusieurs paramètres

L'utilité potentielle de Classes à Plusieurs Paramètres (MPTCs, Multi-Parameter Type Class) a été notée dès le départ [**72, 33**]. Leur nécessité apparaît quand on essaie d'offrir une interface générique pour des collections. Par exemple, pour la fonction générique d'insertion dans une collection `c`, on s'attendrait à la signature polymorphe `insert :: e -> c e -> c e`. Les arbres binaires sont une sorte de collection qui respecte un invariant d'ordre et leur fonction d'insertion prend plutôt le type `binarytree_insert :: Ord e => e -> BinaryTree e -> BinaryTree e`. La contrainte additionnelle serait indésirable dans l'interface de la classe. On conclue donc qu'il serait impossible de définir une instance de collection générique pour les arbres binaires, indépendamment du type d'éléments `e`. Cependant, on pourrait avoir des instances pour les types `e` qui ont un ordre, et on doit donc quantifier

`e` au niveau de la classe, comme second paramètre. Alors, on peut définir l'instance qui ajoute une contrainte sur le second paramètre `e` :

```
instance Ord e ⇒ Coll (BinaryTree e) e where
  insert :: e → BinaryTree e → BinaryTree e
  -- ...
```

Dans la définition partielle ci-haut, `insert` n'a plus besoin de mentionner la contrainte d'ordre, puisque celle-ci est satisfaite au niveau de la classe.

Les MPTCs ont été implantées dans la première publication du langage Gofer en 1991 [**24**]. D'ailleurs, son créateur, Mark P. Jones, les aborde dans ses travaux sur les *qualified types* [**28**]. Une façon de les ajouter à Haskell a été présentée dans l'article de Chen *et al.* sur les *parametric types classes* [**8**]. Cependant, la conception du langage Haskell est demeurée conservatrice en évitant de les inclure dans le standard de 1998 [**54**], malgré leur addition dans GHC en 1997 [**24**]. Pour les ajouter, Peyton Jones *et al.* ont d'ailleurs détaillé un ensemble de considérations et de restrictions à appliquer pour que l'inférence demeure décidable avec cette fonctionnalité [**55**].

L'utilisation des MPTCs a été limitée au départ par le fait qu'il est facile d'écrire des classes trop polymorphes, pour lequelles l'inférence est difficile. En particulier, si le type d'une des fonctions ne mentionne pas tous les paramètres, son application sera ambigue ! Pour répondre à ce problème, Jones a introduit les *functional dependencies* [**31, 32**]. Celles-ci permettent d'indiquer que certains paramètres doivent être déduits à partir de certains autres. En ajoutant une annotation à la classe à l'aide des *functional dependencies*, on définit ce qui se rapproche d'une fonction de type à type. D'ailleurs, cette fonctionnalité a été exploitée pour statiquement faire certains calculs au niveau des types en Haskell [**21**]. Une autre fonctionnalité qui résout les ambiguités des MPTCs est celle des types (et synonymes de types) associés (*associated types and type synonyms*) [**7, 6**]. Cette fonctionnalité étend les définitions de classes en leur permettant non seulement de contenir des valeurs membres, mais également des types ou synonymes de types. On peut ainsi éviter d'ajouter certains paramètres à la classe, et les remplacer par des synonymes associés :

```
class Coll c where
  type Elem c
  insert :: Elem c → c → c
  -- ...
```

D'ailleurs, ces fonctionnalités on plus tard été généralisées pour permettre plus largement de définir des fonctions de type ouvertes [**65**] et puis fermées [**17**], indépendamment de classes. On qualifie une fonctionnalité comme étant "ouverte" si on peut

ajouter des nouveaux cas sans modifier les cas ou les définitions précédentes, et inversement "fermée" s'il faut modifier une définition pour ajouter un nouveau cas. Les classes de types sont intrinsèquement ouvertes, puisqu'on peut toujours ajouter de nouvelles instances. Les familles de types partagent cette propriété centrale, et cette proximité est mise en évidence par le fait que les classes de types peuvent être simulées à l'aide des familles de types [**67**].

(3) Instances chevauchées

Il peut parfois être utile de définir des instances chevauchées. Par exemple, on pourrait imaginer vouloir implanter une version spécialisée d'un algorithme pour un type donné, et pour une classe `C` quelconque, définir à la fois une instance générique pour les listes `C [a]` et une instance spécifique pour les chaînes de caractères `C [Char]`. La seconde instance serait proscrite par le standard, qui impose une syntaxe stricte pour éviter les chevauchements [**54**]. Cependant, vu leur utilité, l'implantation GHC relaxe ces contraintes pour permettre les chevauchements. L'article de Peyton-Jones *et al.* sur l'espace de conception des classes en Haskell détaille les compromis que ça implique [**55**].

2.2. Scala : Paramètres Implicites

Scala est un langage qui mélange la programmation fonctionnelle et orientée-objet, et demeure interopérable avec Java [**49**]. Dès sa deuxième version en 2006, il inclue la fonctionnalité des paramètres implicites, qui y sont utilisés de façon similaire aux classes de types de Haskell [**47**]. Remarquons que les paramètres implicites de Typer sont différents de ceux de Scala : bien que les deux servent à éviter de fournir des arguments de façon explicite, ceux de Typer sont déduits par unification, et on verra qu'en Scala, ils sont plutôt obtenus par une recherche dirigée par les types. Les paramètres implicites de Scala ont été employés de façon assez répandue avant d'être analysés plus formellement comme une alternative aux classes de types dans l'article d'Oliveira *et al.* [**51**]. On note que le mot "classe" sera utilisé dans deux sens : "classe" de types au sens de Haskell, ou "classe" d'objets au sens orienté-objet de Scala/Java. La distinction sera relevée lorsque nécessaire.

2.2.1. Fonctionnement des paramètres implicites

Les paramètres implicites servent avant tout à réduire la répétition dans le code en évitant de passer explicitement certains arguments aux sites d'appels, et d'inférer quelle variable doit être passée selon le type du paramètre.

```
1 object ImplicitParamsDemo extends App {
2     def getImplicitInt(implicit i : Int) = i;
3     implicit val anInt : Int = 4;
```

```
4     println(getImplicitInt); // affiche 4
5 }
```

Dans cet exemple, on définit une fonction `getImplicitInt`, qui prend implicitement un paramètre de type `Int`. On définit ensuite une variable `anInt`, de type `Int`. La variable est définie avec le mot clé `implicit`, qui la rend éligible à être utilisée pour combler les arguments d'appels implicites. On retrouve un tel appel à ligne suivante, à la fonction `getImplicitInt`, dont le résultat est affiché avec `println`. On peut également explicitement passer un argument à certains sites d'appels, en écrivant `getImplicitInt(5)` dans l'exemple précédent.

Les variables qui sont considérées pour un site d'appel donné sont donc celles qui sont déclarées comme implicites dans sa portée locale, et qui sont accessibles par nom. Les paramètres implicites eux-mêmes sont aussi des variables éligibles. Il y a quelques complications dans la définition de la "portée locale" venant de l'aspect orienté-objet de Scala, que nous allons ignorer ici.

Pour chaque argument à combler, il doit y avoir au moins une variable implicite du type correspondant dans la portée, sinon on rapporte une erreur. S'il y en a plusieurs, Scala définit des heuristiques pour sélectionner la plus "spécifique" [**48**]. Une erreur est soulevée si une ambiguïté demeure.

2.2.2. Une alternative aux classes de types

Dans la section 2.1 sur les classes de types de Haskell, on a constaté que la traduction du dictionnaire a principalement deux rôles : traduire les classes et instances en types et données (aggrégeant les composantes d'une interface), et déterminer quelles instances utiliser à chaque site d'appel en suivant les types. L'intuition menant au patron des classes de types en Scala est qu'on utilise directement les "dictionnaires" présents dans le langage, soit les classes et les objets de Scala, et que les paramètres implicites répondent à l'aspect de résolution dirigée par les types.

Par exemple, on suit la traduction du dictionnaire pour exprimer la classe de type `Ord` – des opérateurs de comparaison – avec une classe orientée-objet (ici un `trait`, mais la distinction n'est pas imporante) :

```
1 trait Ord[A] {
2   def lt(x : A, y : A) : Boolean
3   def eq(x : A, y : A) : Boolean
4 }
```

Voyons deux exemples d'instances. D'abord, l'instance pour les entiers est définie par un objet qui implante le `trait Ord[A]`. On rend l'objet disponible pour la résolution avec le mot-clé `implicit` :

```
implicit object intOrd extends Ord[Int] {
  def lt(x : Int, y : Int) = x < y
  def eq(x : Int, y : Int) = x == y
}
```

Le second exemple sert à illustrer l'aspect récursif de la résolution, présent en Scala de la même façon qu'en Haskell. L'instance de `Ord` pour les paires est polymorphe, et pour implanter l'ordre lexicographique, il faut que les éléments de la paire puissent également se comparer. Dans des cas comme celui-ci, la traduction du dictionnaire génère des fonctions, et c'est aussi ce qu'on fait ici : on définit une méthode implicite qui construit le dictionnaire approprié.

```
implicit def pairOrd[A, B](implicit ordA : Ord[A], ordB : Ord[B]) : Ord[(A, B)] =
  new Ord[(A, B)] {
    def lt(x : (A, B), y : (A, B)) =
      if ordA.eq(x._1, y._1) then ordB.lt(x._2, y._2) else ordA.lt(x._1, y._1)
    def eq(x : (A, B), y : (A, B)) =
      ordA.eq(x._1, y._1) && ordB.eq(x._2, y._2)
  }
```

Quand une méthode implicite est sélectionnée pendant la résolution, ses paramètres implicites sont résolus immédiatement de façon récursive. Afin d'assurer la terminaison de ce processus, Scala impose des contraintes sur ces paramètres [**47**].

On peut utiliser `Ord` et ses instances avec le petit exemple de la fonction `min` qui sélectionne le plus petit de deux arguments :

```
def min[A](x : A, y : A)(implicit ordA : Ord[A]) : A =
  if ordA.lt(x, y) then x else y

println(min((3, 5), (3, 3))) // affiche (3,3)
```

On remarque qu'à la différence des autres exemples de la traduction du dictionnaire, le paramètre implicite `ordA` vient après les autres paramètres de `min` : Scala impose que les paramètres implicites apparaîssent en dernier. Cela a d'ailleurs pour conséquence qu'ils ne peuvent pas activement contribuer à l'inférence des arguments qui précèdent, puisque l'inférence des types dans les listes d'arguments procède de gauche à droite [**51**].

On peut rapidement constater de cet exemple qu'il y a plusieurs différences entre ce patron en Scala, et l'utilisation des classes de types en Haskell.

(1) Scala permet d'avoir des instances chevauchées, ce qui n'est pas directement possible en Haskell (sauf avec une extension dans GHC).

(2) Similairement, les MPTCs s'expriment sans extension avec les implicites.

(3) Les instances en Scala peuvent être définies localement, alors qu'en Haskell, leur statut plus global en fait une fonctionnalité moins composable.
(4) Les paramètres implicites sont plus flexibles, puisque la résolution dirigée par les types ne se limite pas qu'aux classes de types, mais peut également être utilisée pour des fins de configuration, par exemple.
(5) En Haskell, les contraintes de classe peuvent être inférées de la définition d'une fonction. À l'opposé, il faut déclarer les paramètres implicites en Scala.
(6) Finalement, l'expression des instances en tant que variables et paramètres en Scala force les programmeurs à les nommer. Cela a ses avantages puisque cela permet également l'application explicite lorsque nécessaire, mais c'est malheureusement plus verbeux.

De récents développements mitigent ce dernier désavantage. Effectivement, l'obligation de déclarer les paramètres implicites de chaque fonction demandait beaucoup de répétition, ce qui a poussé au développement et à l'ajout des *types de fonctions implicites* dans Scala 3, qui est encore au stade de version d'évaluation [**50**]. Les types de fonction implicites donne un statut *first-class* aux fonctions avec des paramètres implicites. De plus, leur usage permet maintenant d'avoir des instances anonymes, auxquelles on peut faire référence avec la fonction `implicitly`, une version plus polymorphe de notre fonction exemple `getImplicitInt` :

```
1 def implicitly[T](implicit x : T) = x
2
3 def min'[A](x : A, y : A) : Ord[A]?⇒ A =
4   if implicitly[Ord[A]].lt(x, y) then x else y
```

Ici, la fonction `min'` est équivalente à `min`, sauf que son argument implicite est anonyme, dénoté par le nouveau type de flèche implicite `?⇒`, et on s'y réfère avec `implicitly`. Ces changements facilitent l'usage des implicites pour les classes de types, mais les autres différences relevées par rapport à Haskell demeurent.

2.2.3. Utilisation répandue

Les paramètres implicites et spécifiquement le patron des classes de types est largement utilisé en Scala. Dans leur article sur les types de fonctions implicites, Odersky *et al.* ont étudié les 120 projets Scala les plus populaires sur Github (selon leur nombre d'étoiles) et on découvert que 84% d'entre eux utilisaient les paramètres implicites [**50**]. Une étude plus large de l'utilisation des implicites s'intitule *Scala Implicits Are Everywhere* (Les implicites de Scala sont partout) [**35**]. Cette étude s'est penchée sur 7280 projets, et recense que 46.2% des sites d'appels impliquent des paramètres implicites, et que 30% des appels implicites correspondent à des classes de types.

2.2.4. Implicites en Ocaml

Les liens entre les classes de types et les modules de Ocaml ont été étudiés par Wehr et Chakravarty, qui ont établit des traductions entre ces deux fonctionnalités qui rendent explicites leurs différences [**74**]. Leur traduction des classes de types se base sur la traduction du dictionnaire et utilise des modules explicitement passés par paramètre.

Dreyer et al. observent aussi que les modules sont déjà utilisés pour configurer explicitement des programmes, et que les classes de types permettent quant-à-elles de faire de la configuration implicite, et proposent d'étendre Ocaml avec les classes de types [**15**]. Les modules leur servent de dictionnaires d'instances, et les signatures de modules correspondent aux classes elles-mêmes. Dans leur article, les contraintes de classes sont inférées sans avoir des instances globalement uniques, ce qui impose d'autres limitations. Par exemple, les auteurs ont décidé de confiner les déclarations d'instances au *top-level* et elles requièrent l'ajout d'annotations de types. Une alternative plus récente s'inspire des implicites de Scala et utilise aussi les modules comme dictionnaires [**76**]. Comme en Scala, les contraintes de classes doivent y être décrites explicitement, ce qui évite des contraintes et simplifie l'usage de la fonctionnalité.

2.3. Propriétés de la résolution

Suivant la présentation de deux visions alternatives des classes de types, incluant leur version plus générale avec les paramètres implicites, on résume certaines de leurs études formelles et les propriétés intéressantes qui en découlent. Ces propriétés serviront également de critères pour évaluer et caractériser les travaux de ce mémoire au chapitre 7.

2.3.1. Haskell : cohérence et unicité globale

En explorant des extensions aux classes de types de Haskell, Peyton Jones *et al.* ont noté certaines contraintes qu'ils jugeaient désirables pour leur conception. Entre autres, ils recherchaient un système où on a la cohérence, une propriété qu'ils définissent comme suit : toutes les dérivations de types possibles pour un programme mènent à une seule sémantique dynamique [**55**]. Autrement dit, s'il y a plusieurs façons de construire un dictionnaire d'instance, elles doivent toutes être équivalentes du point de vue de l'exécution. Cette propriété est utile pour faciliter le raisonnement sur le comportement d'un programme. La cohérence a récemment été prouvée formellement pour un système avec des classes de types et certaines extensions [**4**].

Par ailleurs, on a déjà mentionné que pour des fins d'inférence, les instances doivent être définies globalement en Haskell, et qu'elles ne peuvent pas se chevaucher, sans extension

au langage. Cette unicité globale sert aussi à assurer, par exemple, qu'on n'utilise pas accidentellement des instances distinctes dans l'application de différentes fonctions d'une même structure de donnée, ce qui pourrait contourner des invariants dans du code Haskell existant. Winant et Devriese soulignent que l'application générale de la cohérence et de l'unicité des instances empêchaient l'application explicite de dictionnaires d'instances. Malgré cela, ils sont parvenus à décrire certaines conditions qui permettent de faire des application explicites à des instances locales, sans menacer ces propriétés [**77**].

L'unicité globale des instances en Haskell simplifie la compréhension du code, mais nuit à la modularité. À l'opposé, la possibilité de définir des instances locales et chevauchées est désirable du point de vue de l'expressivité du système mais s'intègre plus difficilement avec la cohérence puisque plusieurs instances peuvent alors être utilisées.

Comme soulevé dans l'article sur le calcul COCHIS, on peut toujours trivialement satisfaire la cohérence en ayant un système déterministe [**64**]. Par contre, si ce système est plus complexe, il ne sera pas nécessairement plus facile de raisonner sur le comportement des programmes. Les auteurs de cet article apportent une autre propriété intéressante et satisfaite par leur calcul : la stabilité. La stabilité garantit que la même instance sera choisie même si l'on effectue des substitutions. On peut imaginer le code fictif suivant :

```
1  class C a where f :: a → Int
2
3  instance C a   where f _ = 1
4  instance C Int where f _ = 2
5
6  g :: a → Int
7  g = f
8
9  example = g (0 :: Int)
```

Ce code est évidemment erroné en Haskell, puisque les deux instance de la classe `C` sont chevauchées, mais sert néanmoins à illustrer la stabilité. On remarque qu'on a omis la contrainte `C a` dans la signature de `g`, ce qui forcerait l'application implicite à l'instance plus générale `C a`. Cependant, on applique ultimement `g` à un entier, et donc si le corps de `g` se faisait incorporer (*inlining* en anglais) dans `example`, on aurait choisit l'instance plus spécifique `C Int`. Une substitution d'égal à égal a causé un changement d'instance et de comportement du code, ce qui est un contre-exemple de stabilité.

2.3.2. Retours en arrière et terminaison de la résolution

Le calcul des implicites $\lambda_{\Rightarrow}$ a été développé pour modéliser les implicites de Scala et les étudier plus formellement dans un article d'Oliveira et al. [**52**], accompagné d'un rapport technique étendu [**13**], et suivi d'une version approfondie [**53**].

La version approfondie de l'article présente deux ensembles de règles de typage alternatifs : un ensemble "ambigu" et un ensemble algorithmique. L'ensemble de règles ambigu n'est pas dirigé par la syntaxe, et donc pas algorithmique, spécifiant la résolution mais pas exactement comment elle peut se faire. Il est très expressif cependant, puisqu'il permet les retour en arrière (*backtracking*). Les retours en arrière désignent la situation où, dans un calcul avec une résolution récursive, on choisit une instance parmis plusieurs au premier niveau de la récursion, mais on constate que les contraintes ajoutées par cette instance ne sont pas satisfaisables. Alors, on "retourne en arrière", et on essaie un autre choix.

Il se trouve que la version ambigue est malheureusement indécidable [**62**]. L'autre version est algorithmique et donc décidable, mais moins expressive, puisqu'elle exclut les retours en arrière [**13**]. Les auteurs ont motivé cette exclusion en indiquant qu'autrement, il y aurait un impact négatif sur la performance ainsi que sur la facilité à comprendre le code. Les même raisons sont soulevées par les auteurs de COCHIS pour leur système et pour Haskell [**64**]. Pour ce qui est de Scala, puisqu'une seule instance doit être sélectionnée à chaque étape récursive, il n'y a pas d'occasion de faire de retour.

Dans sa thèse de maîtrise, Rouvoet étend $\lambda_{\Rightarrow}$ en présentant un ensemble de règles de typage alternatif, dont l'expressivité est intermédiaire aux deux versions d'Oliveira *et al.* et qui demeure décidable [**62**]. Par ailleurs, il étudie les differentes contraintes qui permettent d'assurer la terminaison de la résolution (et donc sa décidabilité). Parmi celles-ci, il y a celle de l'implantation de Scala, qui impose que chaque étape récursive de la résolution fasse décroître une mesure particulière (les méthodes implicites doivent être *contractives*) [**47**]. Une autre option mentionnée par Rouvoet est de fixer une profondeur maximale à la récursion.

2.4. Types dépendants et lois de classes

On a déjà vu que les types dépendants permettent d'encoder et de vérifier des propriétés plus rafinées. Cela s'applique aussi avec les classes de types. Cette section approfondit cette idée. Les sections suivantes détaillent l'implantation des classes de types dans plusieurs systèmes avec types dépendants.

Dans le standard Haskell de 1998, certaines classes de base sont spécifiées avec un ensemble de lois que leurs instances doivent respecter. Par exemple, la classe `Functor` impose les égalités suivantes :

(1) `fmap id = id` – Un foncteur préserve l'identité. Autrement dit, quand on applique l'identité à chaque point d'une "structure", celle-ci ne change pas.

(2) `fmap (f . g) = fmap f . fmap g` – Un foncteur préserve la composition. Cela veut dire que des fonctions passées en arguments successifs à `fmap` peuvent être composées en un seul appel.

Ces lois sont utiles pour préciser le comportement d'une classe de type au-delà de ce que les types des membres de classes peuvent décrire, sauf qu'elles ne sont pas vérifiées statiquement. Au lieu, utilise généralement des tests pour assurer la qualité du code. D'ailleurs, une bibliothèque développée par Jeuring et al. facilite les tests de lois en utilisant les tests basés sur les propriétés [**27**].

Comme le notent Scott et Newton, les types dépendants permettent d'intégrer les lois en exprimant leur preuve comme un membre additionnel de classe [**66**]. Ils remarquent aussi que ces preuves sont souvent répétitives, et proposent d'en automatiser une partie en utilisant des techniques de programmation générique.

Pour compléter l'exemple, les membre correspondants aux lois pour `Functor` auraient des types qui ressemblent aux suivants en Typer, si on utilise `~` pour dénoter l'égalité de deux fonctions à un seul argument, `∘` pour leur composition, et `id` pour la fonction identité :

```
1 preserveId : fmap id ~ id;
2 preserveComp : (f :?b →?c) → (g :?a →?b) → fmap (f ∘ g) ~ fmap f ∘ fmap g;
```

Les types des preuves suivent donc assez directement l'énoncé des lois, et font référence aux autres membres de la classe (`fmap` ici). On donne un foncteur et les preuves de ses lois au chapitre 7.

2.5. Coq

Coq est un ITP qui implante un calcul coeur avec des types dépendants. Deux fonctionnalité séparées servent à y exprimer le polymorphisme *ad hoc*, en faisant différents compromis. Elles sont abordées dans les sous-sections suivantes.

2.5.1. Les classes de types

Sozeau et Oury ont décrit un moyen d'ajouter les classes de types de façon légère en s'appuyant sur d'autres fonctionnalités déjà présentes dans le système : les enregistrements dépendants, les arguments implicites, et la recherche de preuves par tactiques [**69**].

(1) Enregistrements dépendants

Les enregistrements dépendants correspondent simplement à des types inductifs à un seul constructeur. Ils servent de base pour la traduction du dictionnaire. Les classes sont traduites vers des types d'enregistrements et les instances vers des objets enregistrements. À la différence de Scala et Ocaml, les membres de classe peuvent être de n'importe quel niveau (terme, type, *kind*, etc.), puisque comme Typer, Coq est basé sur un PTS. Une autre différence importante apportée par les dépendances est que chaque membre peut se référer aux précédents. Ainsi, une valeur membre être

accompagnée d'une preuve d'une propriété quelconque, énoncée dans le type d'un membre ultérieur.

On peut prendre l'exemple d'une structure simple d'algèbre : le magma unitaire, qui contient une opération binaire avec un élément neutre.

```
Class UnitaryMagma (A : Type) :=
  {
    empty : A;
    append : A → A → A;

    left_neutrality  : forall x, append empty x = x;
    right_neutrality : forall x, append x empty = x;
  }.
```

La définition de la classe exprime les lois de neutralité dans le type de certains membres qui se réfèrent aux membres précédents. On peut maintenant définir une instance pour l'opération `or` sur les booléens.

```
Lemma orb_right_neutrality : forall x, orb x false = x.
Proof. destruct x. all : auto. Qed.

Lemma orb_left_neutrality : forall x, orb false x = x.
Proof. auto. Qed.

Instance orbUM : UnitaryMagma bool :=
  {
    empty := false;
    append := orb;

    left_neutrality  := orb_left_neutrality;
    right_neutrality := orb_right_neutrality;
  }.
```

Les preuves sont exprimées avec des tactiques, dont le détail n'est pas important ici. Les définitions de classe/instance ci-dessus sont presque équivalentes aux définitions d'enregistrements dépendants correspondantes, sauf pour un détail. En Coq comme en Haskell, la définition d'un type d'enregistrement introduit implicitement des méthodes d'accès pour chaque champ. La définition d'une classe introduit aussi de méthodes d'accès, mais l'instance y est un paramètre qui est implicite plutôt qu'explicite.

(2) Paramètres implicites

Similairement à Typer, on retrouve avec Coq les flèches implicites, pour lequelles les arguments sont déduits par inférence, à l'aide de l'unification. Les classes de types de Coq étendent cette fonctionnalité, en essayant de résoudre, par recherche d'instances, les arguments implicites qui n'ont pas été assignés durant l'inférence.

Ainsi, en interrogeant Coq au sujet de `empty`, on constate effectivement qu'à la fois le type `A` et l'instance sont deux arguments implicites :

```
Print Implicit empty.
```

```
     empty : forall A : Type, UnitaryMagma A → A
     Arguments A, UnitaryMagma are implicit and maximally inserted
```

Coq indique aussi qu'ils sont insérés de façon maximale, c'est à dire que dès qu'on a une référence à `empty`, Coq l'applique directement aux arguments implicites. Si on demande le type de `empty` à Coq, on obtient le type résultant de l'application à la seule instance qu'on a défini : celle pour les booléens.

```
Check empty.
```

```
     empty : bool
```

Pour définir une fonction avec des contraintes de classe, on ajoute des paramètres implicites en utilisant la syntaxe des accolades simples autour de l'argument :

```
Definition append_emptys (A : Type) {_ : UnitaryMagma A}
    : append empty empty = empty :=
  left_neutrality empty.
```

Malheureusement, ces paramètres implicites ne peuvent pas être généralisés automatiquement à partir le la définition de la fonction. Ils doivent être explicitement ajoutés à la définition comme les paramètres implicites en Scala.

(3) Recherche par tactiques

Le mécanisme utilisé pour trouver l'instance à insérer repose sur les tactiques. Les tactiques consistent en un ensemble de commandes impératives qui manipulent l'état et les objectifs de preuves, et forment un langage de plus haut niveau d'abstraction pour rendre les preuves plus succintes et plus résistantes aux changements. Un patron fréquent avec les tactiques consiste à définir une table d'indices (*hints*) qui peuvent être essayés à tour de rôle pour tenter de faire avancer la preuve. La tactique prédéfinie pour la recherche d'instance rassemble les instances de la portée courante dans une telle table. Les définitions d'instances l'étendent automatiquement, et il y a moyen d'y ajouter ou d'en enlever des éléments manuellement pour diriger la recherche. Aucune vérification n'est faite pour éviter les chevauchements, et si plusieurs instances correspondent, le choix peut être dirigé par des priorités numériques, mais n'est autrement pas spécifié. Ce manque de déterminisme peut mener à des ambiguïtés problématiques.

La résolution est très expressive puisqu'elle agit récursivement quand les instances choisies introduisent de nouvelles contraintes de classe, et elle peut effectuer des retours en arrière. Cependant, la terminaison de la résolution n'est pas assurée et il peut être embêtant de déboguer des boucles infinies ou des divergences accidentelles dans la recherche. Le quatrième volume des *Software Foundations* contient une introduction approfondie aux classes de type de Coq et résume leurs avantages et inconvénients [**36**].

2.5.2. Les structures canoniques

Les structures canoniques sont une alternative aux classes de type de Coq. Elles permettent à l'utilisateur de diriger directement l'algorithme d'unification dans certaines situations particulières [**39**]. Concrètement, on peut exprimer la classe précédente comme un enregistrement et représenter le paramètre de classe comme un champ, de la façon suivante :

```
1  Structure UM : Type := UnitaryMagma {
2      A : Type;
3      empty : A;
4      append : A → A → A;
5      left_neutrality  : forall x, append empty x = x;
6      right_neutrality : forall x, append x empty = x;
7    }.
8
9  Definition orbUM :=
10   UnitaryMagma bool false orb orb_left_neutrality orb_right_neutrality.
```

Si on utilise les accesseurs sur un magma unifère implicite et dont on sait que type `A` est `bool`, l'algorithme d'inférence se retrouve avec le problème d'unification `A ? = bool`, et ne peut pas déduire quel enregistrement utiliser, puisqu'il pourrait y avoir plusieurs enregistrement pour lesquels `A` est `bool`. Les structures canoniques permettent de donner cette solution au problème d'unification directement avec la commande `Canonical Structure orbUM : UM.`.

Cette fonctionnalité demande une meilleure compréhension de l'unification. De plus, on ne peut pas directement définir d'instances chevauchées ou utiliser les retours en arrière. Cependant, Gonthier *et al.* documentent des patrons de conception qui permettent d'encoder ces fonctionnalités avec les structures canoniques [**19**].

2.6. Agda

Comme en Coq, les classes de type d'Agda se basent sur les enregistrements dépendants. À la différence de Coq, Agda ajoute les *arguments instance*, un nouveau type de flèche séparé des arguments implicites déduits par unification [**14**]. Alors que les arguments implicites sont

notés par des accolades simples (comme ceux de Coq), les arguments instances utilisent des accolades doubles.

Un aspect pratique de cette version est qu'il est possible d'ajouter rétroactivement des classes de types à du code qui utilisait des enregistrements explicites. Comme en Coq, les définitions d'enregistrements définissent des fonctions d'accès pour chaque champ. En Agda, ces fonctions sont définies dans un module du même nom que l'enregistrement, et ce module peut être "ouvert" pour insérer son contenu dans la portée courante. Le module prend l'enregistrement en paramètre, et à l'ouverture, le paramètre est ajouté à chaque membre. Le module peut également être appliqué pour spécialiser les accesseurs à une valeur particulière. Devriese et Piessens ajoutent une nouvelle forme d'ouverture de module, notée `open ClassName {{...}}`, ou les paramètres du module sont transformés en paramètres instances.

Les auteurs notent que l'expressivité des classes de types de systèmes comme Haskell, Scala et Coq est une épée à double tranchant : elles offrent un modèle de calcul alternatif au niveau des types, ce qui peut permettre du code plus complexe que nécessaire. Ils suggèrent donc d'utiliser un algorithme de résolution plus simple, qui ne permet ni la récursion de la résolution, ni les retours en arrière. Ils proposent aussi que toutes les définitions de la portée courante soient des candidats potentiels pour la résolution. La résolution peut alors être dirigée simplement en définissant ou en cachant des variables.

Cette simplicité est élégante, mais l'implantation en Agda a depuis été étendue pour permettre la recherche récursive, et les instances sont introduites avec le mot-clé "instance". Pour assurer la terminaison, une limite de profondeur de récursion est fixée et peut être changée par l'utilisateur [**71**].

Une limitation de cette implantation est qu'il ne doit y avoir qu'une seule solution dans la portée de chaque problème de résolution, autrement l'ambiguïté est soulevée comme une erreur. Pratiquement, cela empêche l'utilisation d'instances chevauchées. Cependant, cela évite le besoin de retours en arrière dans la recherche d'instances.

McBride illustre l'utilité des arguments instances dans un article présentant une implantation générique de collections ordonnées en Agda [**42**]. En général, on n'a besoin que d'une seule preuve d'une proposition donnée, et le détail de la preuve elle-même est moins important que le théorème qu'elle implante. Cela en fait un candidat idéal pour la recherche dirigée par les types qu'offrent les arguments instances. L'auteur remarque que cette fonctionnalité permet de cacher certains détails dans le code.

Chapitre 3

Élimination des types inductifs

Dans ce chapitre, on étudie l'élimination des types inductifs, et particulièrement leur élimination dépendante, qui est nécessaire à l'écriture de beaucoup de preuves. Après une description générale de cette fonctionnalité, on observe son expression dans deux langages avec les types dépendants, Coq et Agda, et on constate certains aspects plus difficiles d'utilisation. On termine le chapitre avec une discussion des types inductifs de Typer, complétant les bases qui seront utiles pour l'ajout de l'élimination dépendante à Typer au chapitre 5.

3.1. Élimination dépendante

Avant tout, on peut clarifier ce qui est entendu quand on parle d'« élimination ». Cette nomenclature issue de la logique associe à chaque sorte de formule des règles d'introduction et d'élimination. Par exemple, les flèches se font introduire par la forme `lambda` et s'éliminent avec des appels de fonction. Pour les types inductifs, les constructeurs servent d'introduction, et l'élimination se fait par filtrage avec la forme `case`. Un type inductif particulièrement simple est celui des booléens, qu'on introduit par les constructeurs `true` et `false`, et qu'on élimine avec la syntaxe spéciale `if_then_else_`. L'expression Haskell `(if b then 1 else 2) :: Int` constitue donc un exemple d'élimination du type de données `Bool`.

Sans types dépendants, le type de retour de chaque branches dans une `case` ou un `if` doit être identique. Dans l'exemple précédent, les deux branches ont le type `Int`, et changer l'une d'elle pour une chaine de caractère causerait une erreur de compilation. Les types dépendant offrent plus de flexibilité, puisqu'ils peuvent mentionner la valeur de `b`. En l'occurence, on pourrait assigner à l'expression `if b then 1 else "two"` le type `if b then Int else String`.

Si on considère les règles de typage, l'élimination non-dépendante des booléens s'exprimerait comme suit.

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash c : \mathsf{Bool} \quad \Gamma \vdash t : A \quad \Gamma \vdash e : A}{\Gamma \vdash \mathsf{if}\, c\, \mathsf{then}\, t\, \mathsf{else}\, e : A}\ \textsc{Bool-E}$$

Le type de chaque branche et le type retour sont identiques. En comparaison, leur élimination dépendante désigne l'admissibilité d'une règle ressemblant plutôt à :

$$\frac{\Gamma, b : \mathsf{Bool} \vdash A : \mathsf{Type} \quad \Gamma \vdash c : \mathsf{Bool} \quad \Gamma \vdash t : A[b := \mathsf{true}] \quad \Gamma \vdash e : A[b := \mathsf{false}]}{\Gamma \vdash \mathsf{if}\, c\, \mathsf{then}\, t\, \mathsf{else}\, e : A[b := c]}\ \textsc{Bool-E-Dep}$$

La première prémisse indique que A est une proposition qui dépend d'une variable booléenne b, et la seconde introduit l'expression booléenne c, qui est la condition qu'on vérifie. Les deux autres prémisses montrent que le type des branches est spécialisé à la valeur de la condition dans la branche, ce qui relaxe l'objectif de la preuve dans chaque cas. Finalement, la conclusion montre que le type d'un `if` peut dépendre de sa condition. En somme, on remarque que si on a une preuve de A pour chaque valeur de b, alors, on a une preuve de A. L'élimination dépendante est donc indispensable pour effectuer des preuves par cas.

3.2. Types inductifs à la Coq

Il serait peu pratique d'avoir à ajouter, pour chaque nouveau type, les axiomes qui permettent d'introduire le type lui-même, ses constructeurs, et d'ajouter aussi les règles de typage et de réduction pour l'éliminer. Pour cette raison, les types inductifs ont été ajoutés comme extension au calcul des constructions, permettant d'ajouter de nouveaux types sans menacer la correction du système. Cela demande cependant des contraintes dans la définition et l'usage de ces types, notamment pour éviter les termes qui seraient infinis, pour assurer la terminaison des fonctions et vérifier qu'elles sont définies pour toutes les combinaisons d'arguments possibles [**12, 56, 11**]. Comme Typer ne vérifie pas encore la terminaison, nous allons négliger ici ces aspects de terminaison et de correction logique.

Les définitions inductives ont été intégrées dans la théorie des constructions inductives, développée dans la thèse de Werner [**75**]. Cette théorie constitue le calcul coeur du système Coq [**3**].

À l'origine, la définition d'un type inductif fournit une liste de paramètres, une liste d'indices, et une liste de constructeurs avec leurs paramètres et leur affectation des indices [**56**]. Par exemple, le type inductif des listes indexées par leur taille s'écrit de la façon suivante en Coq :

```
1 Inductive vec (A : Type) : nat → Type :=
2   | vnil : vec A 0
3   | vcons : forall (n : nat), A → vec A n → vec A (S n).
```

La première ligne déclare le type `vec` avec le paramètre `A` de type `Type`, et un entier naturel comme indice. La différence entre un paramètre et un indice est qu'un paramètre est quantifié universellement au niveau du type et doit apparaître tel quel dans chaque constructeur, alors qu'un indice peut être contraint de différentes façons d'un constructeur à l'autre. Les lignes suivantes définissent chacune un constructeur : `vnil`, la liste vide, affecte `0` à l'indice, alors que `vcons`, un élément de liste, y affecte `S n` – le successeur de `n`, qui est la longueur du reste de la liste.

Une fois le type inductif défini, on peut facilement construire des valeurs du type `vec` en appelant ses constructeurs :

```
1 Check vnil nat : vec nat 0.
```

Ce qui nous intéresse davantage ici, cependant, ce sont les façons d'éliminer des valeurs de notre type inductif. Une première méthode serait d'ajouter une fonction constante pour le principe d'induction du type défini. Justement, Coq ajoute automatiquement de telles constantes pour chaque `Inductive T`, avec les noms `T_rect` et `T_ind` (la distinction entre les deux n'est pas importante pour nos fins). Il se trouve que la règle Bool-E-Dep, mentionnée précédemment, correspond directement à une telle constante, `bool_rect`, dont le type est :

```
1    bool_rect : forall P : bool → Type, P true → P false → forall b : bool, P b
```

Effectivement, les 4 paramètres correspondent aux hypothèses de Bool-E-Dep, et le type de retour, à sa conclusion. On peut appliquer `Bool_rect` similairement à un `if`, et constater qu'on peut y vérifier l'exemple précédent d'élimination dépendante en spécifiant explicitement `P` (on n'applique pas la condition, qui demeure quantifiée) :

```
1 Check bool_rect (fun b ⇒ if b then nat else string) 1 "two".
```

```
1    bool_rect (fun b : bool ⇒ if b then nat else string) 1 "two"
2         : forall b : bool, if b then nat else string
```

Les constantes d'élimination gèrent directement l'aspect récursif en plus des branchements dans le code, ce qui peut simplifier les arguments de terminaison, mais qui est moins pratique pour la programmation. En Coq, les constantes sont dérivées d'un autre mécanisme d'élimination, soit la forme `match` (`case` dans la terminologie Typer) [**3**]. `match` offre une séparation plus pratique de ces deux aspects, en permettant les branchements comme on les fait dans la programmation fonctionnelle. La terminaison se base alors sur le fait que les appels récursifs se font sur des arguments structurellement plus petits à chaque fois. Les

`match` y interviennent alors plus indirectement en déterminant cette relation d'ordre. Subséquement, d'autres moyens ont été ajoutés à Coq pour rendre la programmation plus facile, comme une syntaxe pour définir des fonctions par filtrage [**68**].

3.3. Élimination dépendante en Coq

L'élimination dépendante est directement intégrée dans les travaux développant les types inductifs et dans Coq. Dans l'article de Coquand et Paulin, la règle d'élimination des types inductifs ressemble à la suivante [**12**] (en la traduisant à la notation des règles précédentes) :

$$\frac{P : (x : A) \to \mathsf{Type} \quad f_n : a_{n,1} \to \cdots \to a_{n,k} \to P(c_n\, a_{n,1}\, \ldots\, a_{n,k})}{\mathbf{rec}(f_1, ..., f_n) : (z : A) \to P(z)}\ \textsc{Case}$$

La règle agit similairement à Bool-E-Dep, mais pour la récursion sur des types inductifs au lieu des expressions `case`. La première hypothèse fait de P une proposition dépendante de type `(x : A) → Type` pour un type inductif `A`. Les f_n désignent les corps de chaque branche, et leur type est la flèche qui prend les arguments du constructeur et retourne P appliqué au constructeur.

Le manuel de Coq décrit des règles similaires, omises ici parce qu'elles emploient encore d'autres notations. L'essentiel est qu'on y trouve la règle `Fix`, correspondant à celle de Coquand et Paulin et une règle `Case`, qui agit de façon semblable, mais sans récursion [**3**].

Dans tous les cas, on constate que l'élimination dépendante est directement incluse dans les règles d'élimination des types inductifs.

Pour revenir à l'exemple, en Coq, l'élimination non-dépendante peut s'écrire `if b then 1 else 2`, comme en Haskell, ou bien en utilisant l'expression `match` équivalente :

```
1 match b with
2 | true ⇒ 1
3 | false ⇒ 2
4 end
```

En comparaison, le `match` pour la version dépendante doit explicitement spécifier le type de retour des branches :

```
1 match b return (if b then nat else string) with
2 | true ⇒ 1
3 | false ⇒ "two"
4 end
```

La clause `return <type>` indique le type de retour dépendant de `b` et prend donc le rôle de l'argument `P` de `Bool_rect`.

3.4. Patron du convoi

Malgré la présence de l'élimination dépendante, certains raisonnements engendrent des difficultés surprenantes. On peut considérer la définition partielle suivante, en Coq :

```
Definition f (b : bool) (neq : ~(b = false)) : (b = true) :=
  match b with
  | true ⇒ eq_refl
  | false ⇒ False_rect (false = true) (neq _)
  end.
```

Ce code commence la preuve d'une tautologie au sujet des booléens : si un booléen b n'est pas égal à $faux$, alors il est égal à $vrai$. Pas faux !

La preuve débute par un branchement sur `b`, en utilisant un `match` dépendant. On doit alors fournir deux preuves :

(1) Dans la branche `true`, on doit prouver `true = true`, ce qui se fait directement par réflexivité avec `eq_refl`.

(2) Dans la branche `false`, on doit prouver `false = true`, ce qui risque d'être difficile ! Heureusement, on sait qu'on ne devrait jamais arriver dans la branche, puisqu'un argument nous indique que `b` n'est pas `false`. On a donc une contradiction à laquelle on peut appliquer le principe d'explosion `False_rect` pour obtenir ce qu'on veut, en particulier `false = true`.

La difficulté est que notre contradiction demande une preuve de `b = false`, que Coq ne nous fourni pas directement, malgré que l'intuition suggérerait que dans la branche `false` du `match`, on apprend que `b` et `false` sont équivalents. Malheureusement, Coq ne donne pas directement accès à cette égalité. Quand on laisse un paramètre fictif `_` à la place de cette preuve, Coq indique qu'il n'arrive pas à l'inférer :

```
Error : Cannot infer this placeholder of type "b = false" in
environment :
b : bool
neq : b <> false
```

Alternativement, on pourrait imaginer que dans la branche, cette égalité tient de façon définitionnelle, et ainsi utiliser `eq_refl` à la place de `_`. Malheureusement, ce n'est pas le cas, et Coq indique qu'il n'arrive pas à identifier `b` et `false` :

```
The term "eq_refl" has type "b = b" while it is expected to have type
 "b = false" (cannot unify "b" and "false").
```

La solution est d'appliquer le patron du convoi, un truc folklorique répandu en Coq, que Chlipala a nommé dans *Certified Programming with Dependent Types* [**10**]. Ce patron exploite

l'élimination dépendante en exprimant le lien entre `b` et `false`, par exemple, à travers une dépendance du type de retour du `match` sur `b` :

```
Definition f (b : bool) (neq : ~(b = false)) : (b = true) :=
  match b as b' return (b = b') → (b' = true) with
  | true ⇒ fun _ ⇒ eq_refl
  | false ⇒ fun eq ⇒ False_rect (false = true) (neq eq)
  end eq_refl.
```

On constate l'ajout d'annotations au `match` :

(1) La clause `as b'` introduit l'identifiant `b'` qui sera substitué dans le type de retour par le constructeur de chaque branche. On doit distinguer `b` et `b'` de cette façon pour pouvoir avoir des références à `b` qui ne seront pas substituées.

(2) La clause `return (b = b') → b' = true` indique le type de retour que l'on désire pour le match. Précédemment, on avait un type de retour plus simple `b = true`. Le patron du convoi consiste à retourner une fonction afin d'ajouter les preuves désirées comme arguments. Ici, on ajoute un argument `b = b'` , qui sera raffiné dans les branches à `b = true` et `b = false`.

Ainsi, dans la branche `true`, on ignore la preuve `b = true`, et dans la branche `false`, on peut utiliser `eq : b = false` pour compléter la contradiction. Finalement, on applique la fonction résultant du `match` à une preuve de `b = b`, soit `eq_refl`. Ce patron nous a donc permis de transporter (ou convoyer) une preuve qui fait référence à `b` de l'extérieur du `match` jusqu'à ses branches.

3.5. Élimination des types inductifs en Agda

En Agda, l'élimination des types inductifs se fait à l'aide de fonctions définies par filtrage. Malgré qu'à l'interne, ces définitions sont traduites en arbres de cas, l'utilisateur n'a pas accès à une forme semblable à `case` directement [**46**].

L'exemple précédent fonctionne idiomatiquement sans difficulté en Agda :

```
f : ∀ (b : Bool) (neq : ¬ (b ≡ false)) → b ≡ true
f true neq = refl
f false neq = ⊥-elim (neq refl)
```

Ici, l'analyse du filtrage raffine le type de `neq` et du type de retour, et tous les `b` se font substituer, ce qui fait qu'on peut directement utiliser `refl` pour nos preuves d'égalité.

Bien que cet exemple a une solution satisfaisante, le filtrage n'est pas toujours suffisant : parfois, on veut effectuer des branchements sur le résultat d'une expression quelconque, alors que le filtrage permet seulement les branchements sur les paramètres des fonctions. Quelques options s'offrent aux utilisateurs d'Agda.

(1) On peut définir des fonctions auxiliaires pour effectuer un filtrage et leur passer les expressions qu'on veut en argument. Pour rendre cette idée plus ergonomique, on peut exploiter les fonctions anonymes et définir une fonction `case_of_` (ou `case_return_of_` pour supporter les dépendances) qui approche de la syntaxe familière des `case` (Agda permet la notation mixfixe comme Typer). On poursuit avec le même exemple en branchant sur `b`, mais pas dans la position de paramètre :

```
case_return_of_ : ∀ {a b} {A : Set a} (x : A) (B : A → Set b) → (∀ x → B x) → B x
case x return B of f = f x

f-case : ∀ (b : Bool) (neq : ¬ (b ≡ false)) → b ≡ true
f-case b neq =
  case b return (λ b' → b' ≡ true) of
  λ { true  → refl
    ; false → {! false ≡ true !}
    }
```

Ici, `case_return_of_` est une fonction qui ne fait qu'appliquer l'un de ses arguments à un autre. Elle sert essentiellement à imiter la syntaxe des `case` de d'autres langages, et à rendre explicite le type de retour. La définition par cas s'effectue dans la fonction anonyme qui est introduite avec la forme `λ`, dans laquelle on fait du filtrage comme à l'habitude. Comme dans l'exemple en Coq, on arrive à compléter le cas `true`, mais dans le cas `false`, on doit prouver `false ≡ true`, et on a de nouveau perdu la trace de la contradiction. Effectivement, on n'a plus de lien entre `b`, la valeur examinée par le `case`, et le paramètre de la fonction anonyme, puisque ces deux expressions sont deux arguments indépendants d'un appel à `case_return_of_`.
Pour résoudre ce problème, on peut employer la même méthode qu'en Coq, et convoyer une preuve `b ≡ b'` :

```
f-convoy : ∀ (b : Bool) (neq : ¬ (b ≡ false)) → b ≡ true
f-convoy b neq =
  (case b return (λ b' → b ≡ b' → b' ≡ true) of
   λ { true  _  → refl
     ; false eq → ⊥-elim (neq eq)
     }) refl
```

(2) Les *With-Abstractions* ont été ajoutées à Agda pour permettre le filtrage sur des expressions supplémentaires au paramètres [**41**]. Cette fonctionnalité s'inspire des guardes dans le filtrage en Haskell en ajoutant une clause supplémentaire dans le filtrage. L'exemple artificiel précédent s'exprime simplement :

```
f-with : ∀ (b : Bool) (neq : ¬ (b ≡ false)) → b ≡ true
f-with b neq with b
```

```
...             | true  = refl
...             | false = ⊥-elim (neq refl)
```

Les clauses `with` sont traduites à des appels à des fonctions auxiliaires, en réordonnant les arguments pour substituer la valeur sur laquelle on branche dans autant de types que possible. On peut ainsi éviter plus de convois.

3.6. Le patron *keep* ou *inspect*, une variation du convoi

Parfois, malgré la permutation des arguments engendrée par le `with`, certaines dépendances sur l'expression du `with` demeurent, et il faut un moyen de l'identifier avec le constructeur de la branche.

On peut considérer l'exemple suivant, aussi assez artificiel, où l'on essaye de définir la fonction d'addition sans filtrage sur les paramètres :

```
has-pred : Nat → Bool
has-pred zero = false
has-pred (suc n) = true

pred : (n : Nat) → has-pred n ≡ true → Nat
pred zero ()
pred (suc n) refl = n

{-# TERMINATING #-}
plus : Nat → Nat → Nat
plus m n with has-pred m
plus m n | false = n
plus m n | true  = suc (plus (pred m refl) n) -- erreur pour refl
```

`has-pred` est le prédicat vérifiant qu'un entier naturel n'est pas nul, et `pred` retourne le prédécesseur d'un entier naturel, en prenant une preuve que son argument n'est pas nul. On définit alors `plus` par récursion sur `m`. Malheureusement, la définition de `plus` ne passe pas la vérification de types :

```
   (has-pred m) ≠ true of type Bool
   when checking that the expression refl has type has-pred m ≡ true
```

On doit faire le lien entre `has-pred m` et chacun des cas de façon manuelle. La solution classique est le patron *inspect*, que Stump appelle le patron *keep* [**70**], qui consiste à effectuer un branchement en ajoutant une preuve d'égalité. On doit utiliser la fonction `keep`, qui sert à garder la trace de son argument au niveau des types :

```
keep : ∀{ℓ}{A : Set ℓ} → (x : A) → Σ A (λ y → x ≡ y)
keep x = ( x , refl )

```

```
4 {-# TERMINATING #-}
5 plus : Nat → Nat → Nat
6 plus m n with keep (has-pred m)
7 plus m n | false , _ = n
8 plus m n | true  , hpm≡true = suc (plus (pred m hpm≡true) n)
```

La fonction `keep` retourne une paire dépendante ou le premier élément est son argument, et le second est une preuve que le premier élément est bien l'argument. En l'occurence, dans la branche `true`, on a une preuve de `has-pred m ≡ true`, ce qui permet de compléter la définition.

On peut remarquer que ce patron est un variation du patron du convoi. La traduction du `with` en fonction auxiliaire évite le besoin de transformer le type de retour en fonction, et au lieu d'y ajouter un paramètre supplémentaire directement, on passe la preuve à convoyer comme second élément d'une paire. La fonction `keep` s'occupe de passer `refl` pour construire la preuve d'égalité.

3.7. Types inductifs dans Typer

Les types inductifs de Typer se distinguent de la présentation classique implantée en Coq. En effet, Typer n'offre pas les indices de type directement, mais on peut les remplacer par des paramètres d'égalité [**45**]. D'ailleurs, dans la présentation habituelle, l'égalité est fournie comme un type inductif utilisant un indice [**5**]. Au lieu, Typer implante l'égalité de façon primitive.

Cette façon alternative de gérer les indices et les égalités ressemble à ce qui se fait dans les langages de l'héritage Hindley-Milner comme Haskell ou Ocaml. Ceux-cis se sont vus ajouter des indices de types *a posteriori*, avec les Types Algébriques Généralisés (GADTs, Generalized Algebraic Data Type), qui s'inspirent eux-mêmes des types inductifs dans les langages avec types dépendants [**2, 78, 9**]. Les GADTs peuvent être traduits à des contraintes implicites d'égalités sur les types dans les langages basés sur Hindley-Milner, puisque l'inférence ne s'y applique qu'aux types [**40**]. En Typer, l'usager peut appliquer cette traduction manuellement et utiliser des égalités en paramètre directement. Alternativement, on pourrait définir une macro pour faire cette traduction de manière automatique.

La présence des paramètres de preuves dans les branches des `case` permet un raffinement manuel qui ressemble aux indices. Cependant, le raffinement du type de retour pour l'élimination dépendante n'était pas possible en Typer jusqu'au présent travail. En effet, le type de retour de chaque `case` y est unifié directement avec le type de chaque branche. Le chapitre 5 décrit comment on peut s'adapter à cette restriction pour implanter l'élimination dépendante et comment le faire en évitant le patron du convoi.

Chapitre 4

Extension de Typer avec les classes de types

Au chapitre 2, on a décrit plusieurs approches à l'implantation des classes de type dans un langage de programmation fonctionnelle. Dans les sections suivantes, on décrit d'abord les contraintes qui ont guidé l'implantation de cette fonctionnalité en Typer, puis on explique son implantation en détail, et quelques extensions à la bibliothèque de base.

4.1. Choix de conception

4.1.1. Dictionnaires

On commence par constater que Typer offre les types inductifs comme moyen d'aggréger ensemble plusieurs termes reliés. Les types inductifs seront donc la cible de la traduction du dictionnaire pour Typer.

4.1.2. Résolution locale

Tout d'abord, on veut pouvoir éviter les contraintes de l'implantation des classes de type en Haskell qui ont un aspect global qui semble peu désirable : on veut pouvoir définir plusieurs instances pour une classe et un type donné, par exemple, et pouvoir le faire peu importe où la classe et le type sont définis.

Cela dirige la conception près de celles de Scala, Agda ou Coq, avec leurs arguments "implicites" ou "instances". L'autre avantage de cette approche est qu'on veut pouvoir se référer implicitement non seulement à des instances de classes classiques, mais aussi à des preuves d'égalité, par exemple. La traduction du dictionnaire sera donc faite manuellement par les programmeurs, pouvant être facilitée par l'utilisation de macros. Le mécanisme principal qui doit être offert par le langage est une résolution basée sur les types.

4.1.3. Inférence des contraintes de classe

On cherche à répliquer l'inférence des contraintes de classe de Haskell, et ainsi, d'éviter d'avoir à déclarer explicitement tous les paramètres instances comme en Scala, Coq, ou Agda. C'est une partie importante de la fonctionnalité, puisqu'elle permet, par exemple, de facilement changer le type de configurations implicites, comme il sera démontré au chapitre 7.

4.1.4. Classes *a posteriori* pour les preuves

Les preuves d'égalité en Typer ne correspondent pas à l'intuition de la surcharge d'opérateur des classes de types. Cependant, comme mentionné dans la section 2.6 sur les arguments instances en Agda, McBride a démontré l'utilité de la résolution d'instance pour la recherche de preuves [**42**].

Comme l'égalité est prédéfinie en Typer, on doit pouvoir la déclarer comme classe de type, ou plutôt comme preuve à chercher par résolution, séparément de sa (pré-)définition. De plus, la syntaxe de Typer comporte déjà trois flèches différentes (normales, implicites, et effacées). Il serait préférable d'éviter d'alourdir la syntaxe avec une quatrième option pour les arguments d'instance. À la place, on choisit d'utiliser directement les arguments implicites (effaçés ou non), et d'utiliser leur type pour décider s'ils seront synthétisés par résolution d'instance ou par unification. Ainsi, on permet aux programmeurs de contrôler un ensemble des types qui devront être considérés comme des classes.

Concrètement, on a considéré, dans l'introduction, la fonction `head : (ls : List ?T) → (p : Not (Eq nil ls)) → ?T`. Par désir de simplicité, tous les paramètres étaient explicites ou "normaux". En réalité, on préfère que les preuves soient effaçées à l'exécution, et la flèche pour `p` serait effaçable (et donc implicite). L'implication que les flèches effacées sont également implicites sert notre conception, puisque les flèches effacées sont souvent utilisées pour les preuves, et bien qu'elles sont implicites, les preuves ne peuvent pas généralement être trouvées par unification. On cherche donc une conception qui permet de déclarer, *a posteriori*, que `Not` est une classe de type pour laquelle on veut faire la résolution d'instances, ce qui transforme la fonction `head` en une fonction avec un paramètre d'instance, sans changements pour la définition de `head` elle-même.

L'autre avantage de cette approche est qu'on réutilise les variables d'unification, qui, dépendamment de leur type, peuvent alors être considérées comme des variables d'instance. On nomme les variables d'unification et d'instance "métavariables", par simplicité, pour la suite. Cette réutilisation simplifie la fonctionnalité.

4.1.5. Résolution sans retours

À la sous-section 2.3.2, on a expliqué que dans certains systèmes où la résolution est récursive et ambigüe, les retours en arrière peuvent augmenter son expressivité. On choisit de permettre la récursion, mais d'éviter les retours en arrière, ce qui est un choix assez commun. De cette façon, on suit l'exemple de Haskell, Scala, COCHIS, et Agda. Ce choix est populaire puisqu'il maintient un bon équilibre entre, d'un côté, l'expressivité, et de l'autre, la performance ainsi que la compréhensibilité du code.

Malgré la récursion, on veut assurer la terminaison de la recherche d'instances, pour offrir une interface plus plaisante aux programmeurs. Sans cela, le compilateur pourrait se trouver dans une boucle de résolution infinie, sans indication de la provenance du problème.

4.2. Détails de l'implantation

On peut commencer par remarquer que la résolution des arguments d'instance se fait purement durant l'élaboration. C'est effectivement un concept qui n'apparaît pas du tout dans le calcul coeur. On a vu que l'inférence de types et la généralisation se passent aussi durant l'élaboration. L'interaction entre les deux est importante.

Comme on réutilise les arguments implicites de Typer, on obtient directement l'insertion de métavariables pour les arguments instances qu'on omet. De plus, on a aussi vu dans la section sur l'inférence en Typer (section 1.4) que si une métavariable n'est toujours pas instanciée à la généralisation, elle est remplacée par un paramètre dans la définition courante. Cela correspond directement à l'inférence de contrainte de classe s'il s'agissait d'une variable d'instance.

Par exemple, on peut définir la classe `Num` pour les types de nombres qui ont les opérations `+`, `-`, `*` et `/`. On le fait en définissant un type inductif à un constructeur qui rassemble les membres dans ses champs. On définit également des fonctions d'accès pour ces membres :

```
1  type Num (α : Type)
2    | mkNum (Num_+ : α → α → α)
3            (Num_- : α → α → α)
4            (Num_* : α → α → α)
5            (Num_/ : α → α → α);
6
7  _+_ = lambda num ⇒ case num | mkNum _+_ _ _ _ ⇒ _+_;
8  _-_ = lambda num ⇒ case num | mkNum _ _-_ _ _ ⇒ _-_;
9  _*_ = lambda num ⇒ case num | mkNum _ _ _*_ _ ⇒ _*_;
10 _/_ = lambda num ⇒ case num | mkNum _ _ _ _/_ ⇒ _/_;
```

Les fonctions d'accès comme `_+_` prennent l'instance comme paramètre implicite. Sans rien changer à Typer, on peut constater que l'inférence de contraintes fonctionne déjà, sans instances. En effet, si on déclare `x = (1 : Int) + (1 : Int)`, Typer infère pour `x` le type

`Num Int ⇒ Int`, puisque la métavariable insérée pour l'argument d'instance n'est pas instanciée avant la généralisation de la variable `x`.

On peut déjà également définir des instances pour `Num` comme données du type inductif `Num`, et les utiliser de façon explicite :

```
1 intNum : Num Int;
2 intNum = mkNum (Num_+ := Int_+) (Num_- := Int_-) (Num_* := Int_*) (Num_/ := Int_/);
```

Avec l'instance ci-dessus, l'expression `_+_ (num := intNum) 1 1` est valide et vaut `2`, comme attendu.

Pour avoir des classes de types, il faut donc ajouter :

- Un moyen de déterminer quelles métavariables doivent être obtenues par résolution d'instances.
- Un algorithme pour choisir l'instance appropriée pour chaque métavariable d'instance à un moment entre son insertion et sa généralisation.

Ces ajouts seront décrits dans les sous-sections suivantes.

4.2.1. Ensemble de classes de types

Afin de donner une interface simple pour déterminer quelles métavariables devraient être obtenues par résolution d'instance, on laisse aux programmeurs le choix d'un ensemble local de types qui doivent être considérés comme des classes. Cela évite le besoin d'ajouter un nouveau type de flèche pour les paramètres instance. Pour continuer l'exemple, on ajoute la déclaration `typeclass Num`, ce qui fait que pour le reste de la portée de cette déclaration, `Num` est dans l'ensemble des classes. Les déclarations de classes sont donc locales pour un maximum de flexibilité.

Concrètement, on étend le contexte d'élaboration avec une liste de types qui sont des classes, et on ajoute un jugement `is_typeclass` au système de types, implanté par une fonction du même nom, qui indique si un type donné doit être résolu, dans un contexte d'élaboration donné. De plus, si le type est un appel comme `Num Int`, cette fonction extrait la tête de l'appel, et teste sa convertibilité avec chaque classe. Par exemple, la tête de `Num Int` est `Num`, qui, en l'occurence, est une classe de la liste quand on se trouve dans la portée de la déclaration `typeclass Num`.

4.2.2. Résolution d'instances

En simplifiant un peu, l'objectif de la résolution est de trouver une variable qui a le type demandé dans le contexte. Pour que cela demeure déterministe, on décide de toujours sélectionner la variable la plus "locale", autrement-dit la plus proche dans le contexte, si plusieurs auraient un type correspondant. En réalité, on veut aussi que la résolution soit

récursive, et on doit donc accumuler les contraintes ajoutées par les instances sélectionnées, et les résoudre à leur tour.

Cela peut paraître simple au premier abord, mais quelques complexités se cachent derrière ces phrases. Lors de la résolution, l'élaboration n'est pas terminée, et certains types ne sont pas complètement connus : ils peuvent mentionner des métavariables qui ne sont pas encore instanciées.

Par exemple, dans la définition `f a b = a + a * b`, `a` et `b` ont un type qui est inconnu ; c'est une métavariable `?α : Type` qui sera généralisée, et on désire obtenir le typage `f : (α : Type) ≡> (num : (Num α)) ⇒ (a : α) → (b : α) → α`. On ne peut donc pas simplement utiliser l'unification entre le type désiré et le type de chaque variable du contexte, puisque l'unification avec l'instance `intNum : Num Int` réussirait en instanciant `α` à `Int`, et la fonction prendrait le type `f : Int → Int → Int`. Ce dernier choix serait une sémantique possible pour des classes de types, et il s'agit de celle qui est choisie par les classes de types en Coq, mais elle paraît moins intuitive et pratique. Après tout, la définition de `f` est autrement polymorphe et ne mentionne aucun `Int`. On préfère que la résolution soit dirigée par les types plutôt que le contraire.

D'un autre côté, l'utilisation de l'unification est nécessaire pour des instance polymorphes. Par exemple, on pourrait vouloir définir une instance de `Num` pour les paires de `Num` qui applique les fonctions séparément pour chaque composante de la paire. Cette instance pourrait avoir le type `pairNum : (α : Type) ≡> Num α ⇒ Num (Pair α)`, et on désire que le paramètre implicite `α` se fasse instancier au bon type quand on cherche une instance pour les paires. Si l'on cherchait un `Num (Pair Int)`, `pairNum` pourrait être sélectionnée en instanciant `α` à `Int`, et l'autre paramètre implicite (l'instance `Num Int`) serait comblé par une résolution récursive, pour finalement obtenir le terme `pairNum intNum : Num (Pair Int)`.

À partir de ces exemples, on détermine que la résolution peut procéder en vérifiant essayant, une à une, chaque variable de son contexte – non pas directement par unification, mais plutôt par une nouvelle procédure qu'on appelle l'appariement (*matching*).

Considérons l'appariement du type `Num (Pair Int)` avec l'instance canditate `pairNum : (α : Type) ≡> Num α ⇒ Num (Pair α)` qui se trouve dans le contexte. L'appariement commence par insérer les métavariables pour les flèches implicites de la variable candidate. Dans l'exemple, on insère des métavariables `?α : Type` pour le paramètre de type implicite, et `?num : Num ?α` pour la contrainte d'instance. Le type "candidat" est alors `Num (Pair ?α)`. L'étape suivante consiste a unifier ce type candidat avec le type recherché, mais en prenant soin de n'instancier uniquement que les métavariables insérées pour l'appariement, les autres métavariables sont considérées comme des constantes pour éviter les appariements infondés (comme celui de `intNum` pour la fonction `f`). Dans le cas courant, `?α` est instanciée à

`Int` et l'unification réussit, terminant l'appariement. L'étape récursive de la résolution vérifie éventuellement les métavariables restantes et recommence le processus avec celles qui correspondent à des classes.

On peut comprendre l'objectif de la résolution récursive comme la synthèse d'un arbre d'appels composé de variables du contexte. Dans l'exemple précédent, l'arbre d'appel synthétisé était `pairNum intNum` (négligeant les variables de type). Comme il s'agit d'appels, il paraît normal qu'on fasse l'insertion des métavariable pour les arguments implicites comme celle qui est faite lors de l'élaboration des appels. On réutilise d'ailleurs directement cette fonction du code de l'élaboration.

Afin d'assurer la terminaison de la résolution, on incrémente un compteur à chaque étape récursive d'une résolution donnée et on fixe un seuil au delà duquel une erreur doit être levée. On choisit un seuil par défaut de 100 étapes récursives, estimant que c'est un nombre qui ne devrait pas être atteint par un usage moyen. Le seuil peut évidement être ajusté si nécessaire.

4.2.3. Report de la résolution

Sachant comment déterminer quelles métavariables rechercher par résolution d'instance et comment effectuer cette résolution, il reste à déterminer quand l'exécuter. On a déjà noté qu'il faut qu'elle se passe entre l'insertion de la métavariable et sa généralisation, mais à son insertion, son type risque de ne pas être suffisamment connu.

Prenons l'expression `_+_ 1 1` : l'élaboration obtient d'abord le type de `_+_ : (α : Type) ≡> (num : (Num α)) ⇒ α → α → α` et insère des métavariables `?α : Type` et `?num : Num ?α` pour ses paramètres implicites. À ce moment là, la métavariable `?α` n'est pas encore instanciée à `Int` (il faut attendre le traitement des arguments suivants), et l'appariement avec l'instance `intNum : Num Int` échouerait.

Pire encore, il est possible qu'à leur insertion, certaines métavariables aient un type complètement (mais temporairement) inconnu, et qu'il ne soit même pas possible de déterminer s'il s'agit d'une classe de types à ce moment là ! Prenons la fonction `implicitly : (class : Type) ⇒ (inst : class) ⇒ class`, correspondant à la fonction du même nom en Scala. En élaborant l'application `(implicitly : Num Int)`, on insère la métavariable `inst : ?class` dont le type n'est connu qu'une fois que le type de retour est unifié avec le type attendu. Entre temps, on ne sait pas encore s'il s'agit d'un paramètre instance.

On décide donc de reporter ces décisions au plus tard possible, juste avant la généralisation d'une métavariable, quand on a autant d'information que possible au sujet de son type.

4.2.4. Contrôle des instances

La déclaration `typeclass` est un outil utile pour contrôler où s'applique la résolution, mais ce n'est pas toujours suffisamment précis. Par exemple, on a mentionné, dans le survol de Typer, la fonction `Eq_unerase : (x :?t) ≡> (y :?t) ≡> (p : Eq x y) ≡> Eq x y` qui réifie une égalité effacée. Si `Eq` est une classe de type, alors cette fonction risque de se faire sélectionner à répétition par la résolution sans jamais avancer le problème. Pour cette raison et pour améliorer la performance, on permet aux utilisateurs de Typer d'indiquer quelles variables peuvent être sélectionnées comme instances. Pour ce faire, on ajoute un indicateur booléen pour chaque variable indiquant si elle peut être une instance ou non. La déclaration `instance <liste de variables>` rend les variables mentionnées disponibles pour la résolution, et inversement, `not-instance <liste de variables>` les cache. Un autre indicateur booléen sert à choisir le comportement par défaut pour les nouvelles déclarations de variables. Provisoirement, `bind-instances` indique que les nouvelles variables sont des instances par défaut, alors que `no-bind-instances` indique le contraire.

4.3. Extension de la bibliothèque de base

On présente maintenant deux classes de types qu'on intègre directement dans la bibliothèque de base.

4.3.1. Littéraux polymorphes

En Haskell, l'utilisation d'un littéral d'entier comme `1` est polymorphe et a pour type `Num p ⇒ p`, puisque la classe `Num` y définit la fonction membre `fromInteger`. On peut donc y utiliser les littéraux à n'importe quel type qui définit une instance de `Num`. On répète cette fonctionnalité en Typer, mais en utilisant la classe distincte `FromInteger` pour respecter la séparation des préoccupations. La classe elle-même et ses instances sont assez simples, comme le démontre l'extrait de code suivant :

```
1  type (FromInteger (α : Type))
2    | mkFromInteger (fromInteger : Integer → α);
3  typeclass FromInteger;
4
5  fromInteger = lambda inst ⇒ case inst
6                              | mkFromInteger fromInteger ⇒ fromInteger;
7
8  IntegerFromInteger : FromInteger Integer;
9  IntegerFromInteger = mkFromInteger I;
10
11 IntFromInteger : FromInteger Int;
12 IntFromInteger = mkFromInteger Integer→Int;
```

La classe de type ne contient que la fonction `fromInteger` comme membre, et on lui définit un accesseur prenant l'enregistrement comme un paramètre implicite. L'instance pour les entiers sans borne (`Integer`) utilise la fonction identité (le combinateur `I`) pour son implantation, alors que l'instance pour les entiers représentés par un mot machine (`Int`) utilise la projection `Integer→Int`.

Heureusement, l'élaboration de Typer a été écrite de façon extensible, et le littéral abstait `<lit>` se ferait expandre à l'appel `typer-immediate <lit>`. Par défaut, `typer-immediate` correspond à la constante prédéfinie `##typer-immediate` qui effectue l'élaboration d'un littéral normalement, mais on peut redéfinir cette variable pour intercepter chaque littéral. On redéfinit `typer-immediate` à une macro qui, lorsqu'elle est appliquée à un entier `<lit>`, retourne l'appel `fromInteger (##typer-immediate <lit>)`.

Ces (re-)définitions ont été intégrées à la bibliothèque de base, en plus du changement plus important d'élaborer les littéraux comme des `Integer` plutôt que des `Int`, par défaut.

4.3.2. Monades

Comme en Haskell, les monades ont une importance centrale, puisque l'interaction avec le monde extérieur se fait à travers la monade `IO`. Essentiellement, elles servent à séquencer des actions dans un contexte d'évaluation représenté par un constructeur de type. Leur interface est composée de deux fonctions : `pure` (aussi appelée `return`, parfois) qui construit une valeur sans effet, et `bind`, qui combine des actions en séquence. L'interface peut être exprimée par la classe suivante en Typer :

```
1 type Monad (M : Type → Type)
2   | mkMonad (bind : (a : Type) ≡> (b : Type) ≡>
3                     M a → (a → M b) → M b)
4             (pure : (a : Type) ≡>
5                     a → M a));
6 typeclass Monad;
7 bind = case ?monad | mkMonad bind pure ⇒ bind;
8 pure = case ?monad | mkMonad bind pure ⇒ pure;
```

L'utilisation des monades est simplifiée par une syntaxe spéciale qu'on appelle la syntaxe `do`. On a généralisé l'implantation monomorphe de la syntaxe `do` de la monade `IO` à n'importe quelle monade.

Une difficulté rencontrée est que bien que le mécanisme de classes de types qui est proposé ici supporte les *constructor classes* comme `Monad`, l'inférence de Typer n'est pas encore assez puissante pour un usage pratique. Par exemple, l'élaboration de l'expression `(pure 1 : IO Int)` échoue parce que Typer n'arrive pas à unifier `IO Int` et `?M ?a`. Bien que ce problème d'unification semble simple, plusieurs affectations des métavariables satisferaient l'égalité : `?M = IO` et `?a = Int`, ou bien `?M = λ x → x` et `?a = IO Int`, ou bien encore `?M = λ _ → IO`

`Int` et un `?a` quelconque, etc. Pour éviter le problème, on a donc également ajusté la syntaxe `do` pour permettre une annotation qui spécifie la monade à utiliser.

Chapitre 5

Amélioration de l'élimination des types inductifs en Typer

Il a été expliqué au chapitre 3 que jusqu'à ce travail, Typer ne raffinait pas le type de retour des branches dans les expressions `case`. De ce point de vue, le comportement ressemblait à ce qui peut se faire dans des langages fonctionnels sans types dépendants. Cependant, quand on entre dans une branche, on apprend des choses sur certaines valeurs du contexte. Les types reflètent ce qu'on sait statiquement sur les valeurs dans les programmes qu'on écrit, alors il faut que cette nouvelle information soit représentée au niveau des types.

Par exemple, dans l'expression `case` suivante, quand on entre dans la branche du `cons`, on sait que `l` est équivalent à `cons x xs`, et on peut en utiliser les composantes.

```
1 case (l : List Int)
2 | nil ⇒ 1
3 | cons x xs ⇒ x
```

Intuitivement, dans une branche avec un constructeur c, on apprend que l'expression éliminée doit se réduire à un appel à c. Une façon simple de manifester cette équivalence est de fournir aux programmeurs une preuve d'égalité entre la valeur éliminée et l'appel au contructeur de la branche. Cette preuve peut alors être utilisée ou ignorée. Cette façon de faire a été proposée dans la conception de Typer, mais n'était pas encore présente dans son implantation [**45**].

On évite les changements à la syntaxe en laissant cette preuve d'égalité anonyme. On peut alors y référer soit par son indice de de Bruijn, soit avec une métavariable d'instance. Les classes de types contribuent donc à en faire un changement inobtrusif, car la majorité du code Typer continue de fonctionner sans modification.

5.1. Ajout des égalités dans les branchements

Afin d'ajouter l'égalité dans les branchement, on doit modifier la sémantique statique de Typer, en changeant la règle pour le typage des expressions `case` (et donc aussi son élaboration). Puisque le contexte de chaque branche est légèrement étendu, le changement se répercute dans la sémantique dynamique aussi, donc dans l'évaluation et l'effaçage.

5.1.1. Sémantique statique

Les règles d'élaboration et de typage du `case`, telles qu'implantées, prennent trop d'espace pour les répéter ici. En résumé, l'élaboration unifie, d'une part, le type de l'expression éliminée avec le type résultant du constructeur de chaque branche, et d'autre part, elle unifie le type de retour du `case` avec ceux de chaque branche. La forme `case` est assez complexe, puisqu'elle lie des variables pour chaque paramètre de chaque constructeur, et donc chaque branche a un contexte légèrement différent. La vérification de type est un peu plus simple puisqu'elle évite l'unification – les types étant déjà tous élaborés – mais doit faire les même manipulations pour étendre le contexte de chaque branche.

Par exemple, l'élaboration de l'expression précédente déduit que l'expression éliminée `l` est une liste d'entiers, ce qui correspond également aux constructeurs des branches. De plus, on déduit que le type de retour des branches est `Int`, et c'est aussi le type qu'on attend pour l'expression `case` entière. Le contexte de la branche `cons` est étendu avec les variables `x` et `xs`.

L'essentiel du changement est d'abord d'obtenir le type `t` de la valeur éliminée `e`, puis, pour chaque branche, de construire l'appel du constructeur à ses arguments `c a*`, et le type d'égalité `Eq (t := t) (c a*) e`. Ayant le type d'égalité en main, on étend le contexte de chaque branche avec une variable abstraite et anonyme de ce type. Pour les branches par défaut (dernière branche avec une variable au lieu d'un constructeur), on ajoute une égalité entre cette variable et la valeur éliminée.

Pour l'expression précédente, on ajoute aux contextes des branches `nil` et `cons`, respectivement, les égalités `Eq nil l` et `Eq (cons x xs) l`.

5.1.2. Sémantique dynamique

La sémantique dynamique est réalisée par la normalisation qui est faite pendant la compilation pour faire l'évaluation au niveau des types, et par une fonction d'effaçage, qui traduit le code élaboré et vérifié dans un lambda-calcul strict.

L'effaçage ne nécessite aucune modification, puisque l'égalité ajoutée au contexte est une variable effaçable. Elle est donc automatiquement retirée du contexte avant l'évaluation.

La normalisation au niveau des types demande plus de finesse. Pour réduire un `case`, on doit savoir quel constructeur est appliqué dans le terme éliminé, afin de sélectionner la

bonne branche. Par exemple, si on essaie de réduire le `case` précédent sans pouvoir réduire `l`, on ne sait pas quelle branche choisir, et la normalisation bloque. Si `l` se réduit à `cons y ys`, cependant, on choisit la branche `cons` et on peut y substituer `x` par `y` et `xs` par `ys`. La réduction procède donc en effectuant la substitution des arguments du constructeur par ceux de la valeur éliminée.

L'égalité qu'on ajoute dans le contexte de la branche `cons` a le type `Eq (t := List Int) l (cons x xs)`. Cependant, quand `l` se réduit et qu'on applique la substitution des paramètres, le type devient `Eq (t := List Int) (cons y ys) (cons y ys)`. Comme les deux côtés deviennent identiques, on peut facilement réaliser l'égalité avec le terme `Eq_refl`. Cela fait qu'on ajoute une dernière étape à la réduction : une fois les paramètres substitués, on applique `Eq_refl` à la valeur éliminée et on l'utilise pour substituer la variable d'égalité ajoutée au contexte.

5.1.3. Correction de l'exemple de la fonction `head`

À l'introduction, on a vu un exemple qui paraissait difficile à réaliser sans l'égalité qu'on a ajouté.

```
1 head : (ls : List ?τ) → (p : Not (Eq nil ls)) → ?τ ;
2 head ls p =
3   case ls
4   | cons x xs ⇒ x
5   | nil ⇒ ?; %% Contradiction ! On sait que ls n'est pas nil !
```

On sait, dans la branche `nil`, que `ls` n'est pas `nil`, parce qu'un argument nous en fournit la preuve. Il est donc évident que la branche ne sera jamais exécutée, mais il faut tout de même qu'elle ait le bon type. On peut exploiter la contradition et utiliser le principe d'explosion `exfalso : False → ?a`[1] pour obtenir une valeur de n'importe quel type :

```
1 Not p = p → False ;
2 typeclass Eq ;
3
4 head : (ls : List ?τ) → (p : Not (Eq nil ls)) → ?τ ;
5 head ls p =
6   case ls
7   | cons x xs ⇒ x
8   | nil ⇒ exfalso (p ?) ;
```

Comme mentionné dans le survol, la négation d'une proposition au niveau des types peut être représentée par une fonction vers `False` (équivalent à `Void`). La contradiction elle-même est construite en faisant l'appel de cette négation à la preuve de la proposition, ce qui donne un terme de type `False` qu'on peut éliminer avec `exfalso`. Ici la preuve passée à `p` est une

1. *Ex falso quodlibet* : D'une contradiction, on peut tout déduire.

métavariable `?` de type `Eq nil ls`, et comme `Eq` est une classe de type, cette métavariable sera comblée par résolution d'instance avec l'égalité qu'on a ajouté au contexte de la branche. On néglige ici l'effacabilité de l'instance, qui alourdirait la syntaxe.

On peut remarquer que cet exemple s'ajuste facilement pour implanter la fonction qui nécessitait un convoi en Coq. En effet, en remplaçant les listes par les booléens on peut écrire :

```
f : (b : Bool) → (p : Not (Eq false b)) → Eq true b;
f b p =
  case b
  | true ⇒?
  | false ⇒ exfalso (p?);
```

La fonction s'écrit plus directement en Typer, et on contourne le recours à l'élimination dépendante, et donc aussi au patron du convoi, puisque l'égalité fournie constitue une alternative simplifiée. Dans la section suivante, on décrit comment on retrouve l'élimination dépendante sur la base des égalités dans les `case`.

5.2. Macro pour l'élimination dépendante

On a introduit l'élimination dépendante avec le principe d'élimination des booléen. On commence donc par montrer comment celui-ci s'implante sur la base des égalités ajoutées aux branches des `case` avant de généraliser le principe aux autres types.

Plus précisément, on veut implanter la fonction `bool-elim : (P : bool → Type) → P true → P false → (b → P b)`. On commence par abstraire toutes les variables et brancher sur `b` :

```
bool-elim : (P : bool → Type) → P true → P false → (b → P b);
bool-elim p pt pf b =
  case b
  | true  ⇒? % On doit utiliser pt, mais le type de retour est P b ≠ P true
  | false ⇒? % On doit utiliser pf, mais le type de retour est P b ≠ P false
```

Le problème est que le type de retour est `P b`, identique pour chaque branche, alors qu'on a des preuves qui dépendent des valeurs de `b`. Heureusement, on sait maintenant que `b` = `true` dans la branche `true` et similairement pour la branche `false`. On peut éliminer ces égalité avec `Eq_cast` pour transporter la proposition vers `P b` dans chaque branche de la façon suivante.

```
typeclass Eq;

bool-elim : (P : Bool → Type) → P true → P false → ((b : Bool) → P b);
bool-elim P pt pf b =
  case b
```

```
6   | true  ⇒ Eq_cast (x := true)  (y := b) (f := P) pt
7   | false ⇒ Eq_cast (x := false) (y := b) (f := P) pf
```

La signature et la signification de `Eq_cast` ont été données à la sous-section 1.3.3. Par souci de clarté, les arguments ont été fournis explicitement, sauf l'égalité elle-même, qui correspond au paramètre implicite de `Eq_cast` nommé `p`. Comme c'est un paramètre implicite qui correspond à une classe de type (`Eq`), il est comblé par résolution d'instance, obtenant l'égalité ajoutée dans chaque branche du `case`.

On se rend compte que plus généralement, l'égalité est principalement éliminée avec `Eq_cast`, et qu'elle sert souvent à préciser le type de retour à partir de la valeur éliminée vers le constructeur de la branche. C'est l'essence de l'élimination dépendante et c'est une structure répétitive et régulière qui peut facilement être exprimée par une macro.

On implante donc une telle macro et on reprend la syntaxe de Coq en permettant d'ajouter les annotation `as <var> return <rtype>` aux expressions `case`. Essentiellement, on y décrit comment le type de retour `rtype` varie en fonction de l'expression éliminée, représentée par la variable `<var>`. Ainsi, notre macro permet de simplifier l'exemple précédent :

```
1   typeclass Eq;
2
3   bool-elim : (P : Bool → Type) → P true → P false → ((b : Bool) → P b);
4   bool-elim P pt pf b =
5     case b as b' return P b'
6     | true  ⇒ pt
7     | false ⇒ pf
```

Dans ce code, l'annotation `return P b'` indique que le type de retour du `case` est `P b` (on substitue la variable `b'` par l'expression éliminée `b`), et que le type de retour de chaque branche est spécialisé au constructeur courant : `P true` pour la branche `true` (on substitue `b'` par `true`), et similairement pour l'autre branche.

Ici, bien qu'on implante l'élimination des booléens, la macro `case_as_return_` n'a rien de spécifique à ce type en particulier. Elle s'applique donc généralement pour faire l'élimination dépendante de n'importe quel type.

On permet une simplification additionnelle : si le terme éliminé est lui-même une variable, on peut l'utiliser directement pour exprimer la dépendance, et omettre la clause `as` :

```
1   typeclass Eq;
2
3   bool-elim : (P : Bool → Type) → P true → P false → ((b : Bool) → P b);
4   bool-elim P pt pf b =
5     case b return P b
6     | true  ⇒ pt
7     | false ⇒ pf
```

Ici, le terme éliminé est `b`, soit une variable, ce qui nous permet d'omettre la clause `as`, qui est implicitement complétée à `as b`.

Le chapitre suivant fait bon usage de la macro `case_as_return_`, et apporte donc d'autres exemples plus variés d'élimination dépendante.

Chapitre 6

Decidable, une solution à la cécité booléenne

On commence ce chapitre par un retour sur les booléens en Typer, puis sur le problème de la cécité booléenne. On voit ensuite comment les types dépendants et le travail du chapitre précédent apportent une première solution à ce problème. On développe finalement une solution plus idiomatique des types dépendants en se hissant au niveau des types.

6.1. Booléens en Typer

En Typer, les booléens n'ont pas de statut particulier. Il ne s'agit pas d'un type prédéfini, mais plutôt d'un simple type inductif :

```
1 type Bool
2   | true
3   | false;
```

À la base, on utilise donc la forme `case` pour éliminer les booléens, mais une macro `if_then_else_` permet également la syntaxe familière et se traduit à un `case` : `if <cond> then <t_expr> else <e_expr>` devient `case <cond> | true ⇒ <t_expr> | false ⇒ <e_expr>`.

6.2. Égalités comme solution à la cécité booléenne

Dans l'introduction, on a rapidement expliqué le problème de la cécité booléenne, popularisé par Harper, qui l'attribue à Dan Licata [**22, 23**]. Essentiellement, le problème est qu'une valeur booléenne n'a pas de signification si on ne connait pas sa provenance. On a vu que la personne qui écrit l'expression Haskell `if (isJust m) then f (fromJust m) else d` sait que l'appel à `fromJust m` est sans danger dans la branche `then`, mais comme le système de types ne tient pas compte de la provenance du booléen, on peut accidentellement échanger les branches sans erreur de compilation.

Harper suggère d'éviter les booléen et d'utiliser des types qui portent plus d'information. Dans l'exemple, le type `Maybe` contient l'information nécessaire dans le cas `Just`, et on peut l'utiliser directement pour faire le branchement :

```
maybe :: b → (a → b) → Maybe a → b
maybe d f m = case m of Just v → f v
                        Nothing → d
```

Alternativement, on peut s'assurer d'attrapper l'inversion de branches à la compilation si le système de types arrive à faire le lien entre `isJust m` et sa valeur. Les types d'égalités dans un système avec les types dépendants peuvent remplir ce rôle. Ainsi, la fonction `fromJust` pourrait demander une preuve que son paramètre `m` satisfait l'égalité `true = isJust m`. Cette obligation de preuve est rapidement satisfaite avec l'égalité ajoutée à chaque branche d'un `case`, tel que décrit au chapitre 5. On peut donc traduire le code de l'exemple en Typer (où l'on utilise le type `Option`, équivalent à `Maybe` de Haskell). On omet les définitions de `is-some` et `from-some`, pour simplifier le code :

```
type Option (t : Type)
  | none
  | some t;

is-some : Option?t → Bool;
from-some : (opt : Option?t) → (p : Eq true (is-some opt)) ≡>?t;

maybe d f opt = if (is-some opt) then f (from-some opt) else d;
```

Dans cet extrait, `from-some` demande implicitement une preuve au sujet de `is-some opt`. À l'appel de `from-some`, la preuve est fournie implicitement par résolution, en sélectionnant l'égalité donnée par la branche du `case` qui est généré par l'expansion du `if`. L'inversion accidentelle des branches n'est plus possible, puisque celles-ci fournissent des preuve différentes et une erreur serait soulevée à la compilation.

On constate donc que les types dépendants avec l'élimination dépendante des types inductifs fournissent une première solution à la cécité booléenne. Par contre, les égalités avec des valeurs booléennes ne sont pas agréables à manipuler directement, puisqu'il faut les éliminer explicitement avec des appels à `Eq_cast`. C'est un des éléments qui complique la définition omise de `from-some`. Le reste du chapitre développe une solution alternative : on suggère d'éviter les booléens et le niveau des valeurs, pour les remplacer par des propositions et se hisser au niveau des types.

6.3. Distinction entre booléens et propositions

Dans son article de blogue, Harper rappelle la distinction importante entre les booléens et les propositions [**22**]. On a vu dans la sous-section sur la correspondance de Curry-Howard (1.3.2) que les propositions logiques peuvent être encodées par des types, et que ces types sont habités par leurs preuves pour les propositions qui sont vraies et prouvables. Elles ont donc l'avantage de conserver leur signification (dans le type) en plus de porter plus d'information (leur preuves). Alors que les booléens n'ont que deux valeurs possibles : `true` ou `false`, une proposition peut être vraie si son type est habité, fausse si la négation de son type est habité, ou il se peut qu'elle soit indécidable et qu'on ne connaisse pas d'habitant ni de la proposition ni de sa négation.

Par exemple, on peut définir une fonction qui teste l'égalité de deux booléens et qui retourne `true` si les arguments sont égaux.

```
1 Bool_== : Bool → Bool → Bool;
2 Bool_== a b =
3   case a | true  ⇒ (case b | true ⇒ true  | false ⇒ false)
4          | false ⇒ (case b | true ⇒ false | false ⇒ true );
```

Alternativement, la proposition `Eq (t := Bool) a b` est la formule qui affirme l'égalité des booléens `a` et `b`. Cette proposition est d'ailleurs décidable pour n'importe quelle paire de valeurs `a` et `b`. Si les deux valeurs sont identiques, la propriété est vérifiée par la preuve `Eq_refl`. Si elles sont différentes, il faut prouver sa négation en raisonnant par l'absurde.

```
1 type Not (Prop : Type)
2   | mkNot ((contra : Prop) → False);
3
4 true≠false : Not (Eq true false);
5 true≠false = mkNot (lambda (p : Eq true false) →
6   Eq_cast (p := p) (f := lambda tf → case tf | true ⇒ True | false ⇒ False) ());
```

Tout d'abord, on veut noter le remplacement de `Not` en enveloppant la flèche par un type inductif pour pouvoir l'utiliser comme un classe de types dans le reste du chapitre.

La définition de `true≠false` est construite en supposant l'existance d'une preuve de `Eq true false` et en dérivant une preuve de la contradiction `False`. Pour ce faire, on élimine l'égalité hypothétique `p` avec `Eq_cast` en transportant une preuve triviale de `True` vers `False`. Ce genre de preuve est assez commun, et on l'abstrait par une macro `discriminate`, rappellant la tactique Coq du même nom, ce qui raccourcit la preuve : `true≠false = discriminate true false`.

6.4. Classe **Decidable**

La présence de deux façons d'exprimer des formules logiques apparaît redondante. D'ailleurs, dans la bibliothèque de base de Coq, on retrouve parfois les deux versions avec des noms différents : `≤ ?` est le prédicat booléen de comparaison d'entiers, et `≤` désigne la proposition correspondante. On pourrait essayer d'utiliser des classes pour surcharger ces noms, mais il faudrait une classe pour chaque opérateur surchargé, ce qui ne serait pas satisfaisant. La classe `Decidable` apporte une solution alternative.

En effet, les fonctions retournant des booléens ne permettent d'exprimer que les proposition décidables en décrivant l'algorithme qui les décide directement. En utilisant une classe pour désigner seulement ces propositions décidables, on peut les mettre en bijection avec leur fonctions correspondantes.

Concrètement, une proposition est décidable s'il existe un algorithme qui mène soit à sa preuve ou la preuve de sa négation. La somme de ces deux options est décrite par le type inductif suivant :

```
1 type Decidable (Prop : Type)
2   | yes (p :: Prop)
3   | no  (p :: Not Prop);
```

Son nom indique qu'il s'agit de la classe `Decidable`, mais cela peut sembler surprenant. Jusqu'à maintenant, les types inductifs utilisés pour les classes n'avaient qu'un seul constructeur qui rassemblait les membres dans ses champs, constituant un dictionnaire. La classe `Decidable` semble différente, mais en réalité, son dictionnaire n'aurait qu'une seule fonction sans argument, retournant la donnée décrite ci-haut : `decide : Decidable ?Prop`. La donnée décrite par `Decidable` est équivalente et plus directe.

Pour poursuivre l'exemple, on écrit l'instance qui décide l'égalité des booléens.

```
1  typeclass Eq; typeclass Not;
2
3  eqBoolDec : (a : Bool) ⇒ (b : Bool) ⇒ Decidable (Eq a b);
4  eqBoolDec = lambda (a : Bool) (b : Bool) ⇒
5    case a return Decidable (Eq a b)
6    | true  ⇒ (case b return Decidable (Eq true b)
7               | true  ⇒ yes
8               | false ⇒ no)
9    | false ⇒ (case b return Decidable (Eq false b)
10              | true  ⇒ no (p := discriminate false true)
11              | false ⇒ yes);
```

En premier lieu, on peut remarquer que la structure de la fonction `eqBoolDec` est identique à celle de `Bool_==`. Il s'agit d'expressions `case` imbriquées qui analysent toutes les combinaisons d'arguments. La différence principale est que `eqBoolDec` traite les preuves et

demande donc un peu plus de verbosité. On utilise l'élimination dépendante pour préciser le type de retour dans chaque branche. Dans les deux branches où les arguments sont identiques, la preuve de `yes` est implicitement satisfaite par l'instance `Eq_refl`. La preuve de `Not (Eq true false)` est implicitment satifaite par la définition `true≠false` de la section précédente, puisque `Not` est déclarée comme étant également une classe de types. La preuve de l'autre branche est donnée explicitement. Puisque les deux paramètres de `eqBoolDec` se retrouvent dans le type de retour, ils peuvent être inférés par unification et on les déclare comme implicites.

Avec cette instance, on peut clarifier la correspondance entre les propositions décidables et les fonctions retournant des booléens. Dans un sens, on peut projeter `Decidable` vers `Bool` avec la fonction `Dec→Bool d = case d | yes ⇒ true | no ⇒ false`, qui oublie simplement les preuves. Dans le sens opposé, on peut transformer une fonction retournant un booléen en proposition décidable en appliquant `Eq true`. Par exemple, si la fonction `Nat_==` teste l'égalité d'entiers naturels, `Eq true (Nat_== ?a ?b)` est une proposition décidable, qui utilise l'instance `eqBoolDec`, bien qu'en pratique, on préfèrerait la proposition `Eq (t := Nat) ?a ?b)` avec sa propre instance.

Cette correspondance est expliquée en profondeur dans le manuel *Programming Language Foundations in Agda*, qui décrit le même type inductif pour `Decidable`, sauf qu'il n'y est pas utilisé comme une classe de types [**73**]. On en suit aussi l'exemple en définissant des fonctions (omises ici) pour les combinateurs logiques comme la conjonction, la disjonction et la négation, qui sont des propositions décidables si leurs arguments le sont également.

6.5. Un `If` pour `Decidable`

Imaginons l'ébauche de code suivante, utilisant une fonction `div` qui effectue une division en demandant une preuve que le diviseur n'est pas nul.

```
1  type Nat | Z | S (n : Nat);
2
3  div : (x : Nat) → (d : Nat) → (p : Not (Eq Z d)) ⇒ Nat;
4
5  exemple x d = if (Nat_== d 0) then
6                  default-value
7              else (div x d (p := <manipulations>));
```

On rappelle la définition des entiers naturels `Nat` à la ligne 1, et la ligne 3 donne la signature de `div`. L'`exemple` utilise un `if` pour vérifier la condition et fournir implicitement la preuve `Eq false (Nat_== a Z)` à la branche avec l'appel à `div`. Cependant, il s'agit d'un encodage assez indirect de la non-nullité de `a`, qu'il faudra manipuler avant de la transmettre à `div`, qui nécessiterait plutôt une preuve de type `Not (Eq Z d)`. Si, au lieu de faire un `case` sur une valeur booléene avec `if`, on effectuait un `case` sur l'instance décidant l'égalité qui nous

intéresse `eqNatDec (a := Z) (b := d) : Decidable (Eq Z d)`, la branche du contructeur `no` fournirait directement la preuve de `Not (Eq Z d)` :

```
exemple x d = case (? : Decidable (Eq Z d))
              | yes ⇒ default-value
              | no ⇒ (div x d);
```

L'instance `eqNatDec` s'obtient par résolution d'instances, si bien qu'on n'a qu'à donner la proposition elle-même pour décrire la condition. La preuve du paramètre implicite de `no` est implicitement disponible comme instance et se fait automatiquement sélectionner pour le paramètre de `div`. Ce `case` ressemble beaucoup à celui qui aurait été généré par un `if_then_else_`, sauf qu'il agit sur des valeurs de `Decidable` au lieu de booléens.

À partir de ce `case`, on extrait une macro `If_then_else_` qui abstrait, cette fois, l'énoncé de la proposition au lieu d'une condition booléene. Ainsi, l'expression `If <Prop> then <t_expr> else <e_expr>` devient `case (? : Decidable <Prop>) | yes ⇒ <t_expr> | no ⇒ <e_expr>`. On utilise le `I` majuscule pour différencier cette version, qui demande un type (une proposition) plutôt qu'une valeur (un booléen). On peut encore simplifier l'exemple :

```
exemple x d = If (Eq Z d) then default-value else (div x d);
```

6.5.1. Démonstration : la fonction `absurd`

Une fonction pratique démontre l'expressivité de `If`, des arguments instances, et des types dépendants : la fonction `absurd`.

Bien souvent, on utilise les types dépendants pour restreindre le domaine d'une fonction. Par exemple, au lieu d'avoir une fonction `head` partielle qui échoue ou qui retourne `none` si son argument est la liste vide, on y ajoute un paramètre prouvant qu'elle n'est pas vide. Cette fonction doit quand même faire un `case` sur la liste pour la décomposer, et sa branche `nil` est alors morte mais doit contenir une expression du bon type. Comme on l'a constaté à la sous-section 5.1.3, cette branche a un contexte absurde qui nous permet de déduire une contradiction et ainsi de lui donner n'importe quel type. Similairement, la fonction `nth` qui retourne le n -ième élément d'une liste pourrait demander que n soit plus petit que la longueur de la liste. `nth` devrait alors contenir une branche morte pour le cas absurde où la liste serait vide (n n'ayant aucune valeur plus petite que la longueur 0).

On définit la fonction `absurd` qui automatise les détails de telles branches. En particulier, `absurd` prend en paramètre une proposition statiquement décidable, et contredite par une hypothèse absurde dans le contexte, et en déduit n'importe quoi.

```
absurd : (Prop : Type) ≡> (d : Decidable Prop) ⇒
         (contra : If Prop then (Not Prop) else Prop) ⇒
        ?a;
```

Considérons l'exemple de `nth : (n : Nat) → (l : List ?t) → (p : n < length l) ≡> ?t`, dans le cas absurde ou `n` est l'index de l'élément recherché, `l` est une liste vide et sa longueur `length l` s'évalue donc à 0, malgré la présence de la preuve `p`. On utilise l'expression : `absurd (Prop := n > length l)`. Ce faisant, la proposition se réduit à `n > 0`, et l'instance de `Decidable` pour la comparaison d'entiers est sélectionnée pour le second argument `d`. Le type de l'argument `contra` se réduit en utilisant implicitement l'instance `d`, puisqu'il est exprimé avec notre nouveau `If`. Par définition, `n < 0` est faux et l'instance `d` se réduit à `no`, fournissant une preuve de `Not (n < 0)`. Ainsi, `contra` prend le type de la preuve contradictoire `n < 0`, qui est résolue à `p`. En résumé, notre appel à absurd omet les preuves et s'expand automatiquement à `absurd (Prop := ...) (d := no ...) (contra := p)`, qui utilise les preuves contradictoires de ses arguments pour appliquer `exfalso` et obtenir le bon type de retour. On arrive ainsi à éviter des détails et spécifier certains termes à un plus haut niveau d'abstraction en utilisant les types dépendants, la résolution d'instance, et la métaprogrammation par macros.

6.5.2. Le choix des ifs

Les branches des ifs ne sont pas toutes pareilles. On a vu que `If` et `Decidable` permettent de manipuler `?Prop` et `Not ?Prop` au lieu de `Eq true ...` et `Eq false ...`, ce qui est une nette amélioration ayant le faible coût d'une définition un peu plus complexe pour l'algorithme de décision, qui doit désormais manipuler quelques preuves. Par contre, les exemples montrés utilisaient principalement le type prédéfini `Eq`. En pratique, il faut également définir le type des preuves pour chaque sorte de proposition utilisée, ce qui peut valoir la peine, à condition qu'on ait besoin de manipuler ces preuves dans les branches (déstructurer, composer, etc). Autrement, les booléens peuvent parfois être suffisants.

Malgré l'amélioration du `If`, la branche `else` contient des `Not`, alors qu'il y a souvent de meilleures alternatives. En effet, les valeurs de type `Not ?Prop` demandent du travail puisqu'il faut définir des fonctions à leur introduction et les éliminer en les appliquant. Beaucoup de propositions ont des "contraires" évidents : par exemple, l'imparité est l'opposé de la parité, et on préfèrerait probablement une preuve de `Odd n` au lieu de `Not (Even n)` dans la branche `else`.

Un autre exemple est celui de l'instance pour `Not ?Prop`, pour laquelle le `else` fourni la preuve de `Not (Not ?Prop)` plutôt que simplement `?Prop`, alors que les deux formules sont équivalentes pour des propositions décidables. Cela suggère donc une généralisation

potentielle de `Decidable`, où un membre additionnel indiquerait la proposition contraire à utiliser pour chaque instance.

Remarquons finalement qu'on a suivi l'exemple de Harper en remplaçant les booléens par un type plus expressif. Cependant, au lieu de `Maybe`, qui n'ajoute plus d'information que dans une seule des branches, on utilise `Decidable`, un type plus symétrique où les deux alternatives ne peuvent contenir que des "contraires".

Chapitre 7

Évaluation

On évalue l'aspect principal de ce travail, soit la fonctionnalité des classes de types et des paramètres instances en trois volets. On commence par une discussion qualitative comparant l'approche décrite ici avec les alternatives étudiées au chapitre 2. Ensuite, on vérifie la performance de l'implantation. Finalement, on complète la bibliothèque de base pour offrir plus d'exemples et s'assurer que la fonctionnalité est bien utilisable en pratique.

7.1. Évaluation qualitative

Le premier volet de l'évaluation consiste à situer Typer et ses classes par rapport au cadre établit dans le chapitre 2. On commence par montrer que les paramètres instances permettent d'exprimer les classes qui ont demandé l'ajout d'extensions à Haskell. On examine ensuite notre implantation de la résolution en considérant les propriétés des modèles théoriques de langages. Puis, la possibilité d'exprimer et prouver les lois de classes est illustrée par l'exemple des foncteurs, mettant en évidence l'utilité de la combinaison des classes et des types dépendants. On poursuit en soulignant l'aspect unique (à la connaissance de l'auteur) des implicites qui sont généralisés en Typer. Finalement, on discute plus largement de l'utilisabilité de cette conception.

7.1.1. Extensions de Haskell

Notre approche basée sur une primitive de résolution permet, similairement à celles de Scala, Agda ou Coq, d'exprimer certaines extensions de classes de Haskell.

À la sous-section 4.3.2, on a montré une classe pour les monades, et il s'agit d'une classe de constructeurs de types. On y a également vu que la résolution fonctionne pour ce genre de classes, mais l'implantation de l'inférence doit être améliorée pour qu'elles soient réellement pratiques.

Les MPTCs se traduisent par un type inductif à plusieurs paramètres, et l'appariement basé sur l'unification n'est pas affecté par leur nombre. Par exemple, la classe `Coercible`

`from to` décrit les transformations d'un type `from` à un autre type `to`, et on peut définir une instance généralisant `FromInteger` :

```
1  type Coercible (from : Type) (to : Type)
2    | mkCoercible (from → to);
3  typeclass Coercible;
4  coerce = lambda coercible ⇒ case coercible | mkCoercible coerce ⇒ coerce;
5
6  fromIntegerCoercion = mkCoercible fromInteger;
7
8  example = (coerce (1 : Integer) : Int);
```

L'exemple fonctionne sans erreur, et l'application de `coerce` trouve l'instance `fromIntegerCoercion : FromInteger ?t ⇒ Coercible Integer ?t` sans problème, démontrant qu'on accepte les MPTCs.

Finalement, les instances chevauchées sont autorisées et on emploie le simple critère de la proximité dans le contexte pour choisir celle qui sera choisie, évitant les définitions complexes spécifiant un ordre de priorité, comme le font GHC et Scala.

7.1.2. Propriétés de la résolution en Typer

La section 2.3 a recensé quelques propriétés d'algorithmes de résolution qui fournissent des garanties utiles pour assurer la facilité de raisonner sur les programmes ayant des implicites.

Bien que ces propriétés sont généralement vérifiées sur des modèles de langages plus abstraits, on s'en sert pour situer l'implantation faite en Typer, puisqu'il nous manque un modèle théorique du langage. Cette discussion n'est donc qu'à titre indicatif, et est proposée sans preuves formelles.

(1) Cohérence

La cohérence, désignant le fait qu'une seule interprétation dynamique est possible pour une expression, même si celle-ci aurait plusieurs élaborations ou typages possibles, est facile à satisfaire pour une implantation comme celle de Typer. En effet, l'élaboration et la résolution de Typer sont des programmes déterministes, et il n'y a donc qu'un typage possible pour un code donné, satisfaisant trivialement la propriété. De plus, notre critère choisissant l'instance la plus proche dans le contexte ne laisse aucune place à l'ambiguité dans la résolution. La comparaison n'est pas tout-à-fait juste, cependant, puisqu'un modèle théorique peut faire des simplifications par désir de clarté ou de concision.

(2) Stabilité

L'exemple d'instabilité décrit à la section 2.3 se traduit directement en Typer, qui ne satisfait donc pas cette propriété.

```
type C (a : Type)
  | mkC (f : a → Int);
typeclass C;
f = lambda inst ⇒ case inst | mkC f ⇒ f;

ca   = mkC (lambda _ → 1);
cint = mkC (lambda (_ : Int) → 2);

g : (b : Type) ⇒> b → Int;
g x = f x;

example1 = g (0 : Int);
example2 = f (0 : Int);
```

Deux instances de `C` sont déclarées : la première, `ca`, fonctionne pour tout type `a` et sa fonction membre `f` retourne `1`, et la seconde, `cint`, s'applique aux entiers et retourne plutôt `2`. La fonction `g` s'accompagne d'une signature sans le paramètre d'instance `C b`, ce qui fait que la contrainte pour l'appel à `f` doit être résolue dans son contexte. L'instance pour les entiers échoue l'appariement avec le type demandé `C b`, mais l'instance polymorphe est sélectionnée avec succès. La variable `example1` s'évalue donc à `1`, en utilisant l'instance `ca`. La variable `example2`, en comparaison, retourne `2`, puisqu'on y choisit l'instance spécifique aux entiers. La seule différence syntaxique entre les deux variables est l'incorporation (*inlining*) de `g`, mais c'est suffisant pour changer l'endroit et le type de la résolution.

Bien que l'absence de stabilité pourrait rendre plus difficile le raisonnement au sujet de l'*inlining* du code utilisant des classes de types, le lieu ou se fait la résolution est ultimement déterminé par les types par un algorithme assez simple. Dans l'exemple, on doit explicitement donner une signature sans la contrainte à `g` pour que la résolution s'y fasse. De plus, Typer se trouve en bonne compagnie dans ce choix, comme cette garantie est également absente en Scala, par exemple [**64**].

(3) Terminaison

La terminaison de la résolution peut facilement être vérifiée. D'abord, l'appariement termine dans un temps limité par la grandeur des types qu'on lui passe en argument, et on fait un nombre fini d'appariements par étape récursive (au plus autant d'appariements que d'instances dans le contexte). De plus, le nombre d'étapes récursives est limité par un seuil fixé préalablement. Toutes ces boucles imbriquées sont donc bornées, et on en déduit informellement que l'algorithme de résolution termine.

On peut remarquer que cette terminaison repose cependant sur un seuil fixé arbitrairement. Il est donc possible que pour des exemples suffisamment élaborés, la limite de récursion soit rencontrée et doive être augmentée.

7.1.3. Lois de classes

La combinaison des classes de types avec les types dépendants permet l'expression des lois comme des membres de classe en Typer. Cependant, les preuves ne sont pas toujours nécessaires ou désirables, donc on préfèrera souvent définir une version sans lois, et une extension avec. Afin de démontrer que l'implantation est adéquate pour cet usage, on donne l'exemple des foncteurs et de leurs lois en Typer.

```
type (Functor (F : Type → Type))
  | mkFunctor (fmap : (a : Type) ≡> (b : Type) ≡>
                      (a → b) → F a → F b);
typeclass Functor;
fmap = case ?functor | mkFunctor fmap ⇒ fmap;

type (LawfulFunctor (F : Type → Type))
  | mkLawfulFunctor
       (functor :: Functor F)
       (preserveId : (t : Type) ≡> fmap (F := F) (a := t) id ~ id)
       (preserveComp : (a : Type) ≡> (b : Type) ≡> (c : Type) ≡>
                       (f : b → c) → (g : a → b) →
                       fmap (F := F) (f ∘ g) ~ fmap (F := F) f ∘ fmap (F := F) g);
typeclass LawfulFunctor;
```

La classe `Functor` correspond à celle de Haskell, et ne définit que la fonction `fmap`. La classe `LawfulFunctor` étend un foncteur (le premier membre de la classe) avec les deux lois de la section 2.4. Le type pour ces lois est légèrement compliqué par les difficultés d'inférence pour les constructeurs de types, qui forcent l'ajout d'arguments explicites pour chaque appel à `fmap`. Pour simplifier le reste de l'exemple, on omettra certains de ces arguments explicites.

Un exemple de foncteur très simple est le foncteur identité, un type qui ne fait qu'envelopper une valeur :

```
type Identity (a : Type)
  | mkIdentity (elem : a);
unIdentity i = case i | mkIdentity elem ⇒ elem;

IdentityFunctor : Functor Identity;
IdentityFunctor =
  mkFunctor (F := Identity)
            (fmap := lambda f i → mkIdentity (f (unIdentity i)));
```

L'implantation de `fmap` pour le foncteur identité ne fait qu'appliquer la fonction passée en argument au contenu. Les preuves des propriétés peuvent être faites séparément :

```
_~_ f g x = Eq (f x) (g x);

Identity_preserves_id : fmap (functor := IdentityFunctor) id ~ id;
```

```
Identity_preserves_id x =
  case x return (Eq (fmap id x) x)
  | mkIdentity e ⇒ Eq_refl;

Identity_preserves_comp
  : (a : Type) ≡> (b : Type) ≡> (c : Type) ≡>
    (f : b → c) → (g : a → b) →
    fmap (functor := IdentityFunctor) (f ∘ g) ~
         fmap (functor := IdentityFunctor) f
         ∘ fmap (functor := IdentityFunctor) g;
Identity_preserves_comp f g x =
  case x return (Eq (fmap (f ∘ g) x) ((fmap f ∘ fmap g) x))
  | mkIdentity e ⇒ Eq_refl;
```

L'essentiel des preuves consiste à faire l'élimination dépendante de l'identité qui serait l'argument des appels à `fmap`. De cette façon, les appels à `fmap` peuvent être réduits avec la définition de `fmap` donnée dans l'instance, et les preuves se simplifient à la réflexivité `Eq_refl`. Il ne reste qu'à rassembler ces définitions ensemble dans une donnée pour l'instance `LawfulFunctor Identity` :

```
IdentityLawfulFunctor : LawfulFunctor Identity;
IdentityLawfulFunctor =
  mkLawfulFunctor (F := Identity)
                  (functor := IdentityFunctor)
                  (preserveId := Identity_preserves_id)
                  (preserveComp := Identity_preserves_comp);
```

7.1.4. Implicites généralisés

Les implicites servent à implanter la surcharge et les classes de types, mais aussi plus généralement à obtenir des variables par leur type, permettant, par exemple, la recherche de preuves. Une autre application potentielle qui dépasse le cadre des classes de types "classiques" est celle des *configurations implicites*. Cette application illustre également les bénéfices de la généralisation des contraintes de classe.

On peut imaginer travailler sur un projet logiciel dans lequel on doit ajouter un paramètre de configuration à une fonction située profondément dans l'arbre d'appels. Considérons le code suivant :

```
f1 a   = a + 3;
f2 b c = f1 (b + c);
f3 x   = f2 x 1;
```

La tâche consisterait à remplacer l'addition dans `f1` par une addition modulaire avec un modulo choisit dans `f3`. Pour le faire de façon pure, on pourrait ajouter ce paramètre à chaque fonction entre son utilisation et sa configuration, mais cela peut représenter un travail

considérable et répétitif. À la place, on voudrait pouvoir éviter de changer les fonctions intermédiaires, et écrire ceci :

```
f1 a   = a + 3 mod (from-config "modulo");
f2 b c = f1 (b + c);
f3 x = let mod-config = mkConfig (name := "modulo") 5 in
       f2 x 1;
```

Pour ce faire, on écrit le type `Config`, définissant une configuration comme étant une valeur enveloppée dans un type qui y associe un nom. On ajoute aussi un fonction d'accès pour cette valeur.

```
type Config (t :: Type) (name : String)
  | mkConfig (val : t);
typeclass Config;

from-config name = lambda (config : Config name) ⇒
  case config | mkConfig val ⇒ val;
```

Les configurations sont implicites, donc `Config` est définie comme une classe de types et l'accesseur prend la donnée implicitement. Pour sélectionner la bonne configuration, `from-config` prend son nom en paramètre.

Avec ces définitions, l'appel `from-config "modulo"` insère une métavariable pour la donnée de configuration, et puisque celle-ci n'est pas résolue dans `f1`, elle est généralisée. On obtient donc le typage `f1 : Config (t := Int) "modulo" ⇒ Int → Int`. La configuration implicite est similairement insérée et généralisée dans chaque fonction intermédiaire, et finalement résolue dans `f3`, où on sélectionne l'instance `mod-config`.

La généralisation des contraintes de classes est cruciale pour éviter la modification à `f2`, et établit un avantage des classes en Typer par rapport à Scala, Agda et Coq. Pour ce qui est de Scala 3, Odersky *et al.* ont considéré le problème de la configuration implicite, et leurs types de fonctions implicites allègent le problème en isolant le changement de configurations dans une définition de type [**50**]. Cependant, pour ajouter une configuration, ils doivent toujours modifier les signatures intermédiaires.

7.1.5. Utilisabilité de la solution proposée

On peut considérer plus généralement comment la conception présentée dans les chapitres précédents affecte la programmation en Typer, et si elle répond aux objectifs fixés dans l'introduction.

La surcharge d'opérateurs qui est à l'origine des classes de types est désormais possible en Typer, comme on l'a vu pour les opérateurs de la classe `Num`, et on a même surchargé

les littéraux d'une façon similaire à Haskell. Ce premier objectif est donc atteint, facilitant modestement la concision et la simplicité de certains extraits de code en Typer.

De façon plus importante, notre solution favorise un code plus polymorphe et modulaire, puisque comme en Haskell, on peut définir une fonction sans mentionner le type précis d'un paramètre, en utilisant plutôt les opérateurs de classes, puis ce type et ses instances se font automatiquement généraliser. Les classes servent alors d'interfaces qui facilitent l'abstraction dans le code.

À la différence de Haskell, on évite une conception où les instance sont globales. Cela permet aux arguments instances d'avoir des usages plus variés que les classes de types. Ainsi, comme en Scala, on offre les configurations implicites, et comme en Adga, on offre un mécanisme permettant une recherche simple de preuves. Ces usages augmentent la concision du code Typer, et facilitent l'écriture de preuves. En somme, ces modifications facilitent donc une programmation dans un style plus implicite et flexible que ce qui était possible avant en Typer.

Plus largement, dans le contexte de la programmation en général, notre solution semble être un point intéressant de l'espace de conception. On a mentionné l'aspect unique de la combinaison des argument instances dans un langage avec la généralisation. On a vu à la dernière sous-section que cette combinaison rend possible les configurations implicites. Pour ce qui est de langages avec types dépendants, on a réussi à compléter la fonctionnalité des types inductifs basés sur un type d'égalité prédéfini en facilitant l'accès aux preuves d'égalité par recherche d'instance. Cela nous a permi d'éviter les difficultés du patron du convoi, simplifiant la programmation dans ce type de langage.

Finalement, on a trouvé une solution à la cécité booléenne en proposant d'utiliser la classe `Decidable` pour remplacer beaucoup d'usages des booléens. Notre suggestion d'utiliser le `If` au niveau des types plutôt que `if` sur des expressions booléenes semble être une approche idiomatique des types dépendants. On estime que ce style de programmation pourrait faciliter les preuves et éviter des erreurs de programmation.

7.2. Tests de performance

Quelques benchmarks on été réalisés, afin de vérifier que l'implantation respecte les complexités algorithmiques attendues, que l'élaboration n'est que raisonnablement ralentie en utilisant les arguments instance, et qu'elle n'est pas affectée quand on ne les utilise pas.

7.2.1. Cadre expérimental

Tous les tests de performance on été exécutés sur un ordinateur portable dont les spécifications sont résumées au listage 1. Pour éviter les variations, l'exécution a été faite après

un redémarrage et la fréquence du processeur a été fixée manuellement à son maximum avec la commande : `sudo cpupower frequency-set -g performance`.

```
$ lscpu | grep -E '(cache)|(name)|(MHz)'; cat /proc/meminfo | grep 'Mem*'
Model name :                      Intel(R) Core(TM) i5-2540M CPU @ 2.60GHz
CPU MHz :                         2961.824
CPU max MHz :                     3300.0000
CPU min MHz :                     800.0000
L1d cache :                       64 KiB
L1i cache :                       64 KiB
L2 cache :                        512 KiB
L3 cache :                        3 MiB
MemTotal :        3915528 kB
MemFree :         2045584 kB
MemAvailable :    2798012 kB
```

Listing 1. Données du matériel employé pour les expériences

Tous les *benchmarks* ont été développés à l'aide le la bibliothèque `core_bench`. Cette bibliothèque facilite leur écriture en exécutant à répétition la fonction à chronométrer avec un nombre croissant de tours de boucles, et en effectuant une régression pour estimer le temps moyen par exécution [**26**].

7.2.2. Vérification empirique des complexités algorithmiques

Comme première étape, on cherche à confirmer que certaines parties de l'implantation ont les complexités algorithmiques attendues. On s'attend, par exemple, à ce que la recherche d'une instance demande un temps qui varie de façon linéaire avec sa profondeur dans le contexte. Similairement, on fait l'hypothèse qu'une résolution récursive nécessite un temps linéaire avec le nombre d'appels récursifs, toute autre chose étant égale.

(1) Temps linéaire de la résolution avec la profondeur de l'instance
On vérifie cette première complexité en effectuant, dans un contexte rempli d'instances, des recherches à différentes profondeurs, à répétition. Pour ce faire, on commence par définir le type unitaire indexé par n'importe quelle valeur, et en faire une classe de types :

```
type Lift (t :: Type) (x : t)
  | mkLift;
typeclass Lift;
```

Le seul constructeur de `Lift`, `mkLift`, ne porte aucun paramètre, ce qui le rendrait isomorphe à la donnée `()` de `Unit`, si on oubliait les paramètres du type. Ceux-cis nous permettent de hisser n'importe quelle valeur `x` au niveau des types. C'est d'ailleurs cette qualité qui donne son nom à `Lift`. Ensuite, on bâtit un contexte rempli avec

100 instances de `Lift`, portant chacune leur indice de de Bruijn comme paramètre `x`. Dans ce contexte, on peut alors se référer à une instance précise en spécifiant le type avec l'indice voulu. Finalement, l'expérience consiste à chronométrer le temps de la résolution pour le type `Lift n`, avec des valeurs de `n` qui sont les puissances de 2 entre 1 et 64. Les temps obtenus apparaissent dans le graphe de la figure 1, et les barres d'erreurs (serrées et presque invisibles) correspondent à l'intervalle de confiance à 95% calculé par `core_bench`.

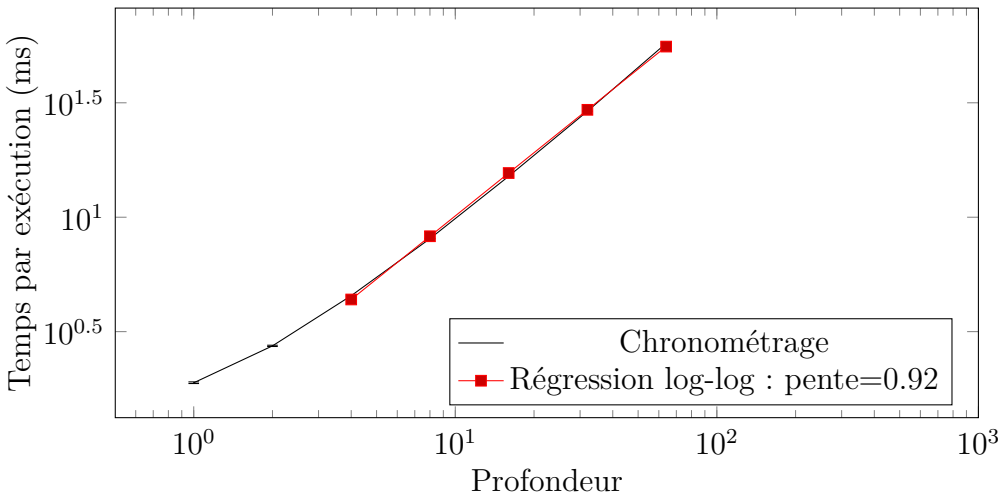

Fig. 1. Chronométrage de la résolution selon la profondeur de l'instance

On utilise un graphe avec des échelles logarithmiques pour estimer la puissance de la fonction obtenue. La courbe des chronométrages correspond presque à une droite sur le graphe, n'étant que très légèrement convexe. On suppose que la convexité pourrait être due à des effet de bord pour des petites valeurs. On exclue donc les deux premier points de la régression linéaire, qui estime alors une pente de 0.92, ce qui est suffisamment près de 1 pour conclure la linéarité de la fonction.

(2) Temps linéaire d'une résolution avec son nombre d'appels récursifs

On procède de façon similaire pour cette seconde complexité à vérifier. On cherche à définir une expérience qui permet facilement de varier le nombre d'étapes récursives de résolution. Pour ce faire, on définit le type des singletons pour les entier naturels, c'est à dire un type `SNat (n : Nat)` qui n'a qu'un seul habitant pour chaque entier naturel `n`.

```
1  type SNat (n : Nat)
2    | Ss ...
3    | Zs ...;
4
```

```
5 Ss' : (n : Nat) ⇒ SNat n ⇒ SNat (S n);
6 Zs' : SNat Z;
```

On omet les détails des constructeurs de `SNat`, puisqu'on y préfère les fonctions `Ss'` et `Zs'` qui servent de constructeurs "intelligents", ayant des types de retour plus précis. On peut constater que le type `SNat n` a la même structure que `Nat`, et que l'unique valeur habitant `SNat n` a la même structure que `n`. `SNat` est donc un type redondant, mais il sert à l'expérience qu'on veut réaliser. Si on en fait une classe de type, la résolution récursive de `SNat n` demande `n` + 1 étapes. En effet, pour `SNat Z`, la première instance trouvée est `Zs'`, qui n'a aucune contrainte, donc la résolution se termine. Pour `SNat (S n)`, l'appariement avec `Zs'` échoue, mais celui de `Ss'` réussit, ajoutant la contrainte `SNat n`, qui poursuit la récursion avec une étape restante de moins.

On effectue donc des résolutions dans un contexte où l'on retrouve les instances `Zs'` et `Ss'`, en demandant des `SNat n` pour des valeurs de `n` qui sont des puissances de 2 entre 1 et 64. Les données obtenues sont illustrées dans la figure 2.

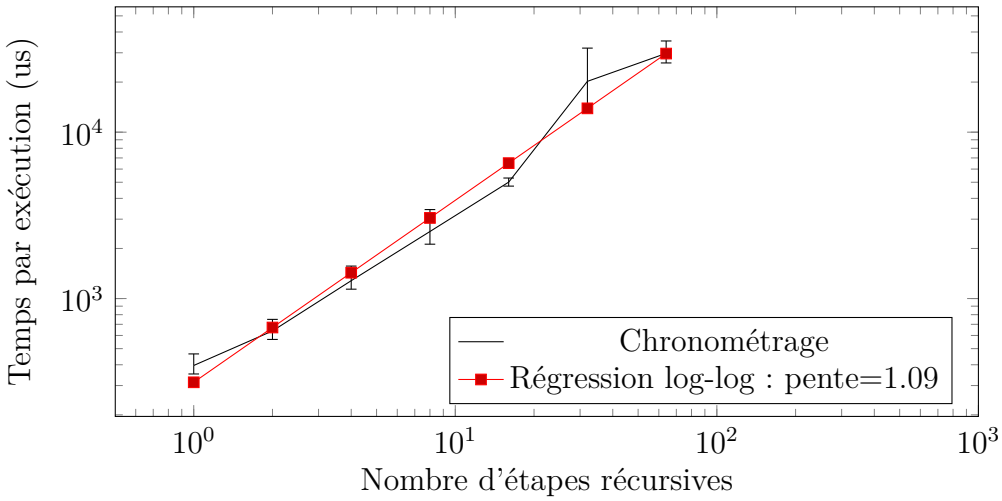

Fig. 2. Chronométrage de la résolution selon le nombre d'étapes récursives

On constate que l'exposant obtenu par la régression est proche de 1, confirmant la complexité presque linéaire de la résolution en fonction du nombre d'étapes récursives.

7.2.3. Ralentissement raisonnable avec arguments instances

On cherche à démontrer que l'élaboration du code qui n'utilisait pas d'arguments instances auparavant, et qui en utilise désormais, n'est que raisonnablement ralenti. Par exemple, on a remplacé les littéraux qui étaient monomorphes par des appels qui demandent l'application à la bonne instance de `FromInteger`. Similairement, on a remplacé la définition

monomorphe `_+_ = Integer_+` par la fonction membre de la classe `Num`. Le prochain *benchmark* consiste donc à effectuer, pour chaque n parmis les puissance de 2 entre 1 et 64, deux versions de l'addition de n littéraux : une avec les classes `Num` et `FromInt`, et l'autre sans. Les résultats sont illustrés au graphe de la figure 3.

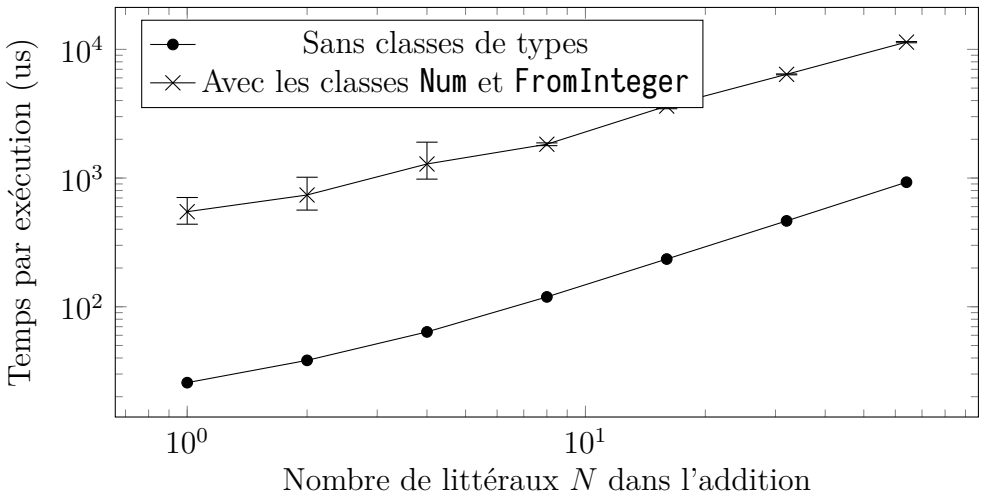

Fig. 3. Chronométrage de l'élaboration d'additions de grandeur N

On constate que la version avec les classes de types s'élabore avec un ralentissement d'environ un ordre de grandeur. Il faut noter qu'il s'agit d'un exemple assez extrême, puisque l'addition sans classes est particulièrement simple à élaborer, et qu'on multiplie la grandeur du terme en ajoutant une instance pour chaque appel de `_+_` et en substituant l'appel `fromInteger (α := ?) (inst := ?) <litteral>` pour chaque littéral. Une part du ralentissement s'explique donc par l'augmentation de la taille du terme élaboré. Puisqu'il s'agit là d'un cas pessimiste, on estime que le temps de résolution est raisonnable en général.

7.2.4. Aucun ralentissement sans arguments instances

Finalement, on veut s'assurer que le code qui n'utilise pas d'arguments instances continue de fonctionner sans pénalité. Pour ce faire, on chronomètre une élaboration avec deux versions de l'interpréteur, soit avec et sans la l'implantation des classes de types. Le code élaboré est cette définition de la fonction calculant les nombres de Fibonacci :

```
1 fib : Int → Int;
2 fib n =
3    case (Int_≤ n 1)
4    | true ⇒ n
5    | false ⇒ Int_+ (fib (Int_- n 2)) (fib (Int_- n 1));
```

Le contenu de la fonction est sans importance, puisqu'on ne s'intéresse qu'aux temps d'élaboration. Notons qu'on élabore `fib` dans un contexte sans entiers polymorphes. Les résultats se trouvent dans le tableau 1, avec les bornes des intervalles de confiance à 95% calculés par `core_bench`.

Tableau 1. Chronométrages d'élaborations de `fib`

Interpréteur	Temps par exécution (ms)	Borne inférieure	Borne supérieure
Sans classes	3.97	-6.68%	+6.01%
Avec classes	3.79	-6.64%	+5.84%

Les temps des deux élaborations sont comparables et on peut conclure que l'ajout de les classes de types ne ralentissent l'élaboration que lorsqu'elles sont réellement utilisées.

7.3. Extension de la bibliothèque de base

Le dernier volet de l'évaluation consiste à écrire davantage d'exemples de classes afin de vérifier que la fonctionnalité est réellement pratiquement utilisable. De plus, il s'agit d'une bonne occasion pour étendre la bibliothèque de base de Typer.

Parmis les exemples de classes en Typer abordés jusqu'à maintenant, on compte `Num`, `FromInt`, `Monad`, `Functor` accompagnée de `LawfulFunctor`, `Coercible`, `Config`, `Decidable`, et les types de preuves `Eq` et `Not`. On ajoute aussi les classes mentionnées dans le cadre d'autres langages dans le reste de ce mémoire, comme la classe `Ord` pour les comparaisons, et l'exemple des magmas unifères est étendu avec des classes pour différentes structures d'algèbre (`Magma`, `Semigroup`, `Monoid`) et leurs instances avec preuves pour `Nat`.

On ajoute encore quelques items à cette liste :

- Les classes `Show a` et `Read a`, inspirées de Haskell, servent à écrire ou lire une donnée de type `a` à partir d'une chaine de caractères. On ajoute également les instances de `Coercible` correspondantes.
- `Functor` décrit seulement les foncteurs dits covariants. On ajoute aussi des classes pour les foncteurs contravariants ou invariants.
- La nouvelle classe `WithDefault` sert à simuler les arguments par défaut. Son type est isomorphe à `Option`, sauf qu'on ajoute une valeur par défaut comme paramètre au niveau des types :

```
1  type WithDefault (α :: Type) (defaultvalue : α)
2    | value α
3    | usedefault;
4  typeclass WithDefault;
5
6  fromWithDefault = lambda (d : WithDefault?defaultvalue) ⇒
7    case d
```

```
8     | value v ⇒ v
9     | usedefault ⇒ ?defaultvalue ;
10
11  example = lambda (x : WithDefault 5) → 1 + (fromWithDefault x) ;
```

En faisant de `WithDefault` une classe de type et en ajoutant `usedefault` comme seule instance pour ce type, on fait en sorte qu'un appel à `fromWithDefault` sur un argument omis retourne la valeur par défaut donnée dans le type. Par exemple, quand on appelle `example` sans argument explicite, l'instance `usedefault` est sélectionnée et `fromWithDefault x` s'évalue à `5`.

À la lumière de cette liste d'exemples, on estime que l'implantation répond aux besoins de la pratique.

Conclusion

La problématique principale à laquelle on a répondu était l'absence de mécanisme de surcharge des noms dans Typer, un langage qui généralise les définitions automatiquement pour offrir le polymorphisme de `let`. Cette dernière qualité en a fait un choix intéressant pour la solution des arguments instances, puisqu'elle a grandement facilité la généralisation des contraintes de classe. Cet aspect distingue la fonctionnalité telle qu'implantée en Typer d'autres langages comme Scala, Coq ou Agda. De plus, on a choisi d'éviter l'unicité globale des instances de Haskell, optant plutôt pour des instance locales comme en Scala. À la différence de la conception originale de Scala, les instances peuvent rester anonymes. Un exemple utile d'instance anonyme, dans le contexte des types dépendants, est celui des preuves, qu'on préfère souvent ne pas nommer, parce que leur type est déjà assez descriptif. La recherche dirigée par les types constitue alors un excellent moyen de s'y référer.

On a choisi d'avoir une résolution d'instances aussi simple que possible, en lui faisant essayer les instances du contexte dans leur ordre d'apparition, sans établir d'ordre de priorité secondaire comme on peut en trouver dans GHC, Coq, et Scala. Cette quête de simplicité apparait aussi dans notre décision d'éviter les retours en arrière et d'assurer la terminaison en limitant les résolutions récursives par un nombre arbitraire et ajustable. Ces décisions facilent le débogage quand il est nécessaire.

Le fait que Typer ait des types dépendants a aussi permis l'expressions de classes avec des lois vérifiées statiquement. Pour écrire les preuves de ces lois, il a cependant fallu ajouter l'élimination dépendante des types inductifs, en passant par des égalités dans les branches des expressions `case`. Heureusement, l'utilisation de ces égalités anonymes a en retour été facilitée par les arguments instances.

En complétant la conception originale de Typer avec les types inductifs basés sur un type d'égalité prédéfini, on a réussi à éviter les difficultés qui on engendré le patron du convoi en Coq, également présent en Agda (où il est parfois appelé *keep* ou *inspect*).

Par ailleurs, la combinaison de des fonctionnalités ajoutées à Typer a ouvert la porte à une solution élégante au problème de la cécité booléenne, et on a proposé un moyen d'éviter les booléens et les `if` en les remplaçant par des propositions décidables et une nouvelle forme `If`. En plus d'éviter la cécité booléenne, ce nouveau `If` facilite l'accès au preuves

intéressantes dans les branchements. Cet exemple de la classe `Decidable` et bien d'autres ont servi à étendre la bibliothèque de base de Typer, augmentant son utilité.

Références bibliographiques

[1] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand et H. Abelson : Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9) :26–76, septembre 1998.

[2] Lennart Augustsson et Kent Petersson : Silly type families, 1994.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi et Benjamin Werner : The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, mai 1997. Projet COQ.

[4] Gert-Jan Bottu, Ningning Xie, Koar Marntirosian et Tom Schrijvers : Coherence of type class resolution. *Proc. ACM Program. Lang.*, 3(ICFP), juillet 2019.

[5] Pierre Castéran et Eduardo Gimenez : A Tutorial on [Co-]Inductive types in Coq, 2004. Révision complète d'une présentation des constructions inductives et coinductives en Coq.

[6] Manuel M. T. Chakravarty, Gabriele Keller et Simon Peyton Jones : Associated type synonyms. *In Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, page 241–253, New York, NY, USA, 2005. Association for Computing Machinery.

[7] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones et Simon Marlow : Associated types with class. *In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005. Association for Computing Machinery.

[8] Kung Chen, Paul Hudak et Martin Odersky : Parametric type classes. *In Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, page 170–181, New York, NY, USA, 1992. Association for Computing Machinery.

[9] James Cheney et Ralf Hinze : First-class phantom types. Rapport technique, Cornell University, 2003.

[10] Adam Chlipala : *Certified Programming with Dependent Types : A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[11] Thierry Coquand : Pattern Matching with Dependent Types. *In Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[12] Thierry Coquand et Christine Paulin : Inductively defined types. *In* Per Martin-Löf et Grigori Mints, éditeurs : *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[13] Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee et Kwangkeun Yi : Extended report : The implicit calculus, 2012.

[14] Dominique Devriese et Frank Piessens : On the bright side of type classes : Instance arguments in agda. *SIGPLAN Not.*, 46(9) :143–155, septembre 2011.

[15] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty et Gabriele Keller : Modular type classes. *In Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 63–70, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Joshua Dunfield et Neel Krishnaswami : Bidirectional typing. *CoRR*, 2019.

[17] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones et Stephanie Weirich : Closed type families with overlapping equations. *In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 671–683, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Taylor Fausak : 2019 State of Haskell Survey results. https://taylor.fausak.me/2019/11/16/haskell-survey-results/, novembre 2019. Accessed : 2020-11-07.

[19] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski et Derek Dreyer : How to make ad hoc proof automation less ad hoc. *In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 163–175, New York, NY, USA, 2011. Association for Computing Machinery.

[20] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones et Philip L. Wadler : Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2) :109–138, mars 1996.

[21] Thomas Hallgren : Fun with functional dependencies. *In Proc. of the Joint CS/CE Winter Meeting*, 2000.

[22] Robert Harper : Boolean blindness. https://existentialtype.wordpress.com/2011/03/15/boolean-blindness/, mar 2011. Accessed : 2020-11-13.

[23] Robert Harper : *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd édition, 2016.

[24] Paul Hudak, John Hughes, Simon Peyton Jones et Philip Wadler : A history of haskell : Being lazy with class. *In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery.

[25] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain et John Peterson : Report on the programming language haskell : A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5) :1–164, mai 1992.

[26] Roshan James : Core_bench : better micro-benchmarks through linear regression. https://blog.janestreet.com/core_bench-micro-benchmarking-for-ocaml/, mai 2014. Accessed : 2021-04-01.

[27] Johan Jeuring, Patrik Jansson et Cláudio Amaral : Testing type class laws. *In Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 49–60, New York, NY, USA, 2012. Association for Computing Machinery.

[28] Mark P. Jones : A theory of qualified types. *Science of Computer Programming*, 22(3) :231–256, 1994.

[29] Mark P. Jones : Functional programming with overloading and higher-order polymorphism. *In* Johan Jeuring et Erik Meijer, éditeurs : *Advanced Functional Programming*, pages 97–136, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[30] Mark P. Jones : A system of constructor classes : overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1) :1–35, 1995.

[31] Mark P. Jones : Type classes with functional dependencies. *In* Gert Smolka, éditeur : *Programming Languages and Systems*, pages 230–244, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[32] Mark P. Jones et Iavor S. Diatchki : Language and program design for functional dependencies. *In Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, page 87–98, New York, NY, USA, 2008. Association for Computing Machinery.

[33] Stefan Kaes : Parametric overloading in polymorphic programming languages. *In* H. Ganzinger, éditeur : *ESOP '88*, pages 131–144, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[34] M. S. Krishnan, C. H. Kriebel, Sunder Kekre et Tridas Mukhopadhyay : An empirical analysis of productivity and quality in software products. *Manage. Sci.*, 46(6) :745–759, juin 2000.

[35] Filip Křikava, Heather Miller et Jan Vitek : Scala implicits are everywhere : A large-scale study of the use of scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA), octobre 2019.

[36] Leonidas Lampropoulos et Benjamin C. Pierce : *QuickCHick : Property-Based Testing In Coq*. Software Foundations series, volume 4. Electronic textbook, août 2018. Version 1.0. http://www.cis.upenn.edu/~bcpierce/sf.

[37] Xavier Leroy : Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1) :42–54, janvier 2006.

[38] Zengyang Li, Qinyi Yu, Peng Liang, Ran Mo et Chen Yang : Interest of defect technical debt : An exploratory study on apache projects. *In Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020.

[39] Assia Mahboubi et Enrico Tassi : Canonical Structures for the working Coq user. *In* Sandrine Blazy, Christine Paulin et David Pichardie, éditeurs : *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 de *LNCS*, pages 19–34, Rennes, France, juillet 2013. Springer.

[40] Conor McBride : Type inference needs revolution. Workshop on Type Inference and Automated Proving, 2015.

[41] Conor Mcbride et James Mckinna : The view from the left. *Journal of Functional Programming*, 14(1) :69–111, 2004.

[42] Conor Thomas McBride : How to keep your neighbours in order. *In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 297–309, New York, NY, USA, 2014. Association for Computing Machinery.

[43] S. McConnell : An ounce of prevention. *IEEE Software*, 18(3) :5–7, 2001.

[44] Robin Milner, Robert Harper, David MacQueen et Mads Tofte : *The Definition of Standard ML (Revised)*. MIT Press, 2nd édition, mai 1997.

[45] Stefan Monnier : Typer : Ml boosted with type theory and scheme. *Journées Francophones des Langages Applicatifs*, pages 193–208, 2019.

[46] Ulf Norell : *Towards a practical programming language based on dependent type theory*. Thèse de doctorat, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[47] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman et Matthias Zenger : An overview of the scala programming language (2. edition). Rapport technique LAMP-REPORT-2006-001, EPFL Lausanne, Switzerland, 2006.

[48] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman et Matthias Zenger : The scala language specification (version 2.13), 2019.

[49] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman et Matthias Zenger : An overview of the scala programming language. Rapport technique IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[50] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller et Sandro Stucki : Simplicitly : Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), décembre 2017.

[51] Bruno C.d.S. Oliveira, Adriaan Moors et Martin Odersky : Type classes as objects and implicits. *SIGPLAN Not.*, 45(10) :341–360, octobre 2010.

[52] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee et Kwangkeun Yi : The implicit calculus : A new foundation for generic programming. *In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 35–44, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi et Philip Wadler : The implicit calculus : A new foundation for generic programming. 2014.

[54] Simon Peyton Jones : *Haskell 98 Language and Libraries : The Revised Report*. Journal of functional programming : Special issue. Cambridge University Press, 2003.

[55] Simon Peyton Jones, Mark Jones et Erik Meijer : Type classes : an exploration of the design space. *In Haskell workshop*, January 1997.

[56] Frank Pfenning et Christine Paulin-Mohring : Inductively defined types in the calculus of constructions. *In* M. Main, A. Melton, M. Mislove et D. Schmidt, éditeurs : *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag.

[57] Benjamin C. Pierce : *Types and Programming Languages*. The MIT Press, 1st édition, 2002.

[58] Benjamin C. Pierce et David N. Turner : Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1) :1–44, janvier 2000.

[59] Baishakhi Ray, Daryl Posnett, Vladimir Filkov et Premkumar Devanbu : A large scale study of programming languages and code quality in github. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 155–165, New York, NY, USA, 2014. Association for Computing Machinery.

[60] Didier Rémy et Jérôme Vouillon : Objective ml : A simple object-oriented extension of ml. *In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 40–53, New York, NY, USA, 1997. Association for Computing Machinery.

[61] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric et Zachary Tatlock : Qed at large : A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3) :102–281, 2019.

[62] Arjen Rouvoet : Programs for free : Towards the formalization of implicit resolution in scala. Mémoire de D.E.A., 2016.

[63] RTI International : The economic impacts of inadequate infrastructure for software testing : Final report. Rapport technique, National Institute of Standards and Technology, 05 2002.

[64] Tom Schrijvers, Bruno C.D.S. Oliveira, Philip Wadler et Koar Marntirosian : Cochis : Stable and coherent implicits. *Journal of Functional Programming*, 29 :e3, 2019.

[65] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty et Martin Sulzmann : Type checking with open type functions. *In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery.

[66] Ryan G. Scott et Ryan R. Newton : Generic and flexible defaults for verified, law-abiding type-class instances. *In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Alejandro Serrano, Jurriaan Hage et Patrick Bahr : Type families with class, type classes with family. *In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, page 129–140, New York, NY, USA, 2015. Association for Computing Machinery.

[68] Matthieu Sozeau : Equations : A dependent pattern-matching compiler. *In* Matt Kaufmann et Lawrence C. Paulson, éditeurs : *Interactive Theorem Proving*, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[69] Matthieu Sozeau et Oury Nicolas : First-Class Type Classes. *Lecture Notes in Computer Science*, août 2008.

[70] Aaron Stump : *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016.

[71] The Agda Team : Agda User Manual : Release 2.6.1.3, février 2021.

[72] P. Wadler et S. Blott : How to make ad-hoc polymorphism less ad hoc. *In Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[73] Philip Wadler, Wen Kokke et Jeremy G. Siek : *Programming Language Foundations in Agda*. juillet 2020. Version 20.07. http://plfa.inf.ed.ac.uk/20.07/.

[74] Stefan Wehr et Manuel M. T. Chakravarty : Ml modules and haskell type classes : A constructive comparison. *In* G. Ramalingam, éditeur : *Programming Languages and Systems*, pages 188–204, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[75] Benjamin Werner : *Une Théorie des Constructions Inductives*. Theses, Université Paris-Diderot - Paris VII, mai 1994.

[76] Leo White, Frédéric Bour et Jeremy Yallop : Modular implicits. *Electronic Proceedings in Theoretical Computer Science*, 198 :22–63, dec 2015.

[77] Thomas Winant et Dominique Devriese : Coherent explicit dictionary application for haskell. *In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 81–93, New York, NY, USA, 2018. Association for Computing Machinery.

[78] Hongwei Xi, Chiyan Chen et Gang Chen : Guarded recursive datatype constructors. *In Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 224–235, New York, NY, USA, 2003. Association for Computing Machinery.