Classes de types de première classe

Matthieu Sozeau[1] et Nicolas Oury[2]

[1] Univ. Paris Sud, CNRS, Laboratoire LRI, UMR 8623, Orsay, F-91405
INRIA Saclay, ProVal, Parc Orsay Université, F-91893
sozeau@lri.fr
[2] Universit de Nottingham
npo@cs.nott.ac.uk

Résumé Les classes de types ("*Type Classes*") ont remporté un grand succès dans le langage de programmation fonctionnelle Haskell et l'assistant de preuve Isabelle, comme solution permettant de surcharger des notations et spécifier avec des structures abstraites en quantifiant sur les contextes. Nous présentons un plongement superficiel des classes de types dans une théorie des types dépendants qui fait des classes des objets de première classe et supporte directement les extensions les plus populaires de Haskell. L'implémentation du système est légère et s'appuie sur des constructions existantes du langage qui sont simplement raffinées pour obtenir un ensemble bien intégré à l'environnement Coq. On présente sur des exemples comment ce système peut être utilisé pour la programmation et la preuve.

La surcharge est un concept de haut-niveau orthogonal aux paradigmes de programmation, bien qu'il soit au centre de la plupart des langages à objets. En termes généraux, la surcharge permet de dénoter à l'aide d'un seul nom un ensemble d'objets (méthodes, fonctions, valeurs) et comprend un mécanisme de désambiguation permettant de résoudre la référence vers un objet particulier. Ce mécanisme peut être la résolution au moment de l'exécution comme dans les langages à objets, ou au moment de la compilation comme pour les classes de types. Les classes de types ont été introduites dans le langage de programmation fonctionnel Haskell pour rendre le polymorphisme *ad-hoc* moins *ad hoc* [1].

On oppose cette forme de polymorphisme au polymorphisme *paramétrique* qui permet de définir des fonctions génériques sur tout type mais empêche d'obtenir des comportement différents selon le type effectif utilisé. Prenons par exemple le type polymorphe $\forall\ \alpha, \alpha \to \alpha \to \mathsf{bool}$ des fonctions à deux arguments retournant un booléen. Il est impossible de construire une fonction de ce type qui pour tout α, déciderait de l'égalité de deux objets de type α. Par exemple, il est impossible de décider de l'égalité sur le type $\mathbb{N} \to \mathsf{bool}$ des prédicats sur les naturels. En revanche il est possible de décider de l'égalité sur un grand nombre de types (booléens, naturels, entiers bornés, listes etc...) et l'on aimerait pouvoir dénoter par le même symbole l'ensemble de ces égalités, en utilisant l'information de typage sur les éléments comparés pour désambiguer la surcharge. Les classes de types permettent ceci *via* une définition de *classe* Eq pour l'égalité contenant une *méthode* surchargée == :

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

```
instance Eq Bool where
  x == y = if x then y else not y
```

Une fois la classe déclarée on peut construire ses instances, comme l'égalité sur les booléens ci-dessus. Il est dès lors possible d'utiliser le symbole == pour dénoter l'égalité sur les booléens dans notre code. On peut aussi écrire des fonctions polymorphes supposant une instance de classe sur le type paramétrique et de la même façon utiliser les méthodes surchargées. Par exemple, pour définir le prédicat polymorphe in qui teste l'appartenance d'un élément à une liste, on a besoin de pouvoir tester l'égalité sur le type des éléments ce qu'on exprime par une contrainte Eq $a \Rightarrow$:

```
in : Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y  : ys) = x == y || in x ys
```

Dans l'article [2], nous montrons comment traduire cette construction dans Coq, en codant les classes par des structures de première classe : les enregistrements dépendants. La traduction utilise le système d'arguments implicites et une procédure de recherche de preuve qui sont implémentés à l'extérieur du noyau pour permettre d'écrire le code utilisant la surcharge et résoudre les contraintes générées automatiquement.

On montre aussi comment interpréter des constructions plus complexes sur les classes comme les concepts de superclasse et sous-classe qui permettent de créer des hiérarchies de structures. On étend l'interprétation très facilement en utilisant toute la puissance du produit dépendant [3] qui permet de spécifier aisément la paramétrisation donc le partage de structures.

À l'aide des types dépendants et en s'alliant au système de tactiques à la base du "*shell*" interactif de Coq, les classes de types permettent aussi de créer des tactiques génériques et personnalisables par l'utilisateur. On peut par exemple définir une classe surchargeant l'ensemble des preuves de réflexivité du système :

```
Class Reflexive (A : Type) (R : relation A) := reflexive : ∀ x : A, R x x.
```

Cette classe est indexée à la fois par un type A mais aussi une valeur R de type relation A (soit une fonction a deux arguments dans `Prop`). On peut instancier cette classe sur l'égalité de Leibniz pour n'importe quel type A :

```
Instance eq_refl : Reflexive A (eq A) := reflexive x := refl_equal x.
```

De même, on peut créer une instance pour l'équivalence logique $\leftrightarrow$:

```
Instance iff_refl : Reflexive Prop iff.
```

Il devient maintenant possible d'utiliser la méthode surchargée reflexive là ou l'on attend une preuve de réflexivité de l'égalité, de l'équivalence ou de toute autre relation déclarée réflexive par l'utilisateur. On a démontré l'utilité d'un tel système pour re-développer la tactique de réécriture généralisée du système Coq [4, chapitre 9].

Le système en est encore à ses premiers pas même s'il fait déjà partie intégrante de Coq 8.2 [5]. En particulier, la recherche de preuve est très grossière puisqu'elle ne permet pas de détecter les ambiguïtés lorsque plusieurs instances sont possibles. Ce trait n'est pas gênant lorsqu'on recherche une preuve mais beaucoup plus lorsqu'on programme et qu'on voudrait par exemple savoir qu'une occurrence de + a deux interprétations possibles dans le même contexte.

Conclusion Pour résumer, nous montrons comment interpréter les "*Type Classes*" dans une théorie des types avec types dépendants. Notre interprétation s'étend aux constructions de haut-niveau des "*Type Classes*" comme les superclasses et donne aussi lieu à des usages originaux grâce aux types dépendants. Nous avons implémenté ce système dans Coq et démontré son utilité sur des exemples variés. Les classes de types dépendantes semblent un outil prometteur pour spécifier, prouver et organiser les développements en Coq.

Références

1. Wadler, P., Blott, S. : How To Make *ad-hoc* Polymorphism Less *ad hoc*. In : ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas. (1989) 60–76
2. Sozeau, M., Oury, N. : First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds. : Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 278–293
3. Oury, N., Swierstra, W. : The Power of Pi. In : ICFP '08 : Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2008) 39–50
4. Sozeau, M. : Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)
5. Sozeau, M. : Type Classes. In : Coq 8.2 Reference Manual. INRIA TypiCal (2008)