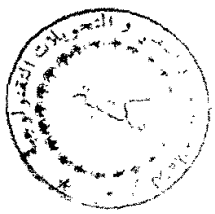
THEORIE DES TYPES ET PROGRAMMATION

Bernard ROBINET

N° 79-21

Avril 1979



Théorie des types et programmation

B. Robinet

Institut de Programmation, Université Pierre et Marie Curie.
LA 248 "Informatique Théorique et Programmation

Résumé : La plupart des langages de programmation permettent d'exprimer des calculs sur diverses classes ou types d'objets. A des fins de correction, tant à la compilation qu'à l'exécution, on doit se demander, pour un programme syntaxiquement correct, si les opérandes et opérateurs sont compatibles ; le contrôle de type permet de répondre partiellement à cette question.

Nous décrivons ici un contrôle statique de types (compile-time checking) qui est une large extension de celui défini par Morris et développé par Ledgard ; traduisant tout programme sous forme de λ-expression, nous associons à cette dernière un type dans le cadre d'une théorie de la fonctionnalité convenable.
Le problème de la correction statique d'un programme est alors ramené à celui de l'association d'un type à une λ-expression.

Abstract : Most programming languages treat computations over different classes of objects or types. For correct compilation and execution, the following question arises : is a program correct so that its operators and operands are compatible ; type checking gives a partial answer.

We describe here a compile time checking (contrôle statique de types) which is a large extension of the Morris system and of the Ledgard's type checker ; mapping any programme into a λ-expression, we associate a type within a suitable theory of functionality. The problem of the static correctness of the program is resolved by associating a type to a λ-expression.

Introduction

Les langages de programmation classiques, tels ALGOL 60, Pascal ou LISP manipulent diverses sortes d'objets : par exemple, une variable ALGOL 60 peut dénoter un nombre, une procédure, une étiquette ; un objet LISP est un atome ou une liste, etc. Le partitionnement de l'univers de discours est souvent appelé système de types, chaque élément de la partition étant nommé type ; c'est ainsi que nous dirons que l'ensemble des entiers naturels a pour type N et que les opérations binaires +,-,x ont pour type $N \times N \rightarrow N$.

Etant donnés deux objets appliqués l'un à l'autre, la question alors se pose de savoir si leurs types sont compatibles. Plus généralement, il y a-t-il pour un programmme donné syntaxiquement correct, violation des restrictions imposées par le système de types du langage : les processeurs de langage-compilateurs ou interpreteurs-répondent à cette question en mettant en oeuvre un contrôle de type.

Ce contrôle est souvent fait à l'exécution du programme, et ce, pour deux raisons. La première est que, lors de cette phase, tous les objets d'un programme ont (ou doivent) avoir leurs "valeurs" déterminées ; si c'est le cas, il est alors facile, à toute demande d'emploi d'un objet de vérifier si sa "valeur" est en accord avec son type. La seconde est imposée par de facheuses particularités des langages de programmation ; ainsi, on peut vérifier à la compilation la correction de l'expression X ÷ Y en ALGOL 60, langage où les types de X et de Y doivent être déclarés ; il est toutefois impossible de vérifier, lors de cette phase, si l'expression (X ↑ Y) ÷ Y conduit à une violation de types étant entendu que le résultat de l'évaluation de X ↑ Y est de type entier ou réel.

Cette façon de contrôler les types des objets, que nous qualifierons de dynamique, présente deux inconvénients majeurs : le premier est dans le considérable ralentissement des performances d'exécution d'un programme, le second étant la non-vérification des branches de programme non exécutées.

Réalisé lors de la phase de compilation, le contrôle statique a pour vocation d'éliminer ces deux ennuis : beaucoup d'informations sont connues durant le déroulement de cette phase, aussi peut-on les utiliser pour répondre, du moins partiellement, à des questions d'ordre sémantique comme la validité des types, et ce, indépendamment de toute exécution des programmes.

Reste à définir avec précision ce que sont les types et leur contrôle dans un langage de programmation. Depuis les travaux de Landin, Böhm, McCarthy et quelques autres, on privilégie le système formel nommé λ-calcul comme modèle des langages de programmation ; Morris [9] puis Ledgard [8], reprenant les travaux de Curry sur la fonctionalité [2], ont défini le type -ou mode- d'un objet informatique comme le caractère de fonctionnalité de la λ-expression modèle de l'objet. Le problème du contrôle statique de type pour un programme donné se ramène alors à celui du calcul de type, s'il existe, de la λ-expression modèle du programme.

Toutefois, si la théorie de la fonctionnalité au sens de Curry est riche de résultats, elle est insuffisante en ce sens qu'elle restreint considérablement la puissance du λ-calcul : cette théorie repose en effet sur l'hypothèse de stratification forte de l'univers laquelle exprime le fait qu'un objet possède un type et un seul ; du coup, beaucoup de λ-expressions, et donc beaucoup de programmes pour autant licites, ne possèdent pas de type.

Afin d'affaiblir cette hypothèse trop restrictive et tenant compte des structures de données, Nolin [10], Robinet [11], Ruggiu [14] puis Ermine et Ressouche [4] ont proposé une extension de la théorie classique de la fonctionnalité permettant d'associer un type à un plus grand nombre de λ-expressions ; bien que le théorème de Sanchis sur l'existence d'une forme normale pour toute λ-expression typée [16] ne soit plus valable, cette théorie reste décidable au sens où il existe un algorithme d'affectation de type, obtenu par composition de deux algorithmes, l'un de calcul proprement dit, l'autre de semi-unification raffinant les types calculés à partir d'une base de types, analogue de la partie déclaration d'un programme.

Nous présentons ici une description synthétique de cette théorie et son application à la définition d'une méthode de contrôle statique de types en programmation.

Après un bref rappel sur le λ-calcul et la théorie classique des types, nous montrons dans une deuxième partie comment construire des modèles en λ-calcul de structures de contrôle et de données usuelles -le lecteur trouvera de plus amples développements dans divers articles ou rapports cités dans la bibliographie- ; dans une troisième partie, nous décrivons sommairement une théorie étendue des types puis nous montrons comment associer un type aux modèles de structures ; enfin, dans une dernière partie nous présentons les deux méthodes utilisables pour associer un type à tout programme