SPECIFICATION DES PROGRAMMES A ISOMORPHISMES DE TYPES PRES

Marcel FOUDA NDJODO
Département d'informatique
Faculté des Sciences - Université de Yaoundé I
B. P. 812 Yaoundé
Cameroun

Abstract: To reduce the gap between syntax and semantics of types, a new style of specification "up to isomorphisms" of types is considered. In this style of specification, built-in rewriting rules identify equivalent datas, i.e. datas isomorphic in the structure of categories defining the semantics of the language. Programs written "up to isomorphisms" of types are more flexible since they accept any data isomorphic to a data of the domain. This paper gives the theoretical framework where the problem can be correctly formalised.

Mots-clés: Spécification algébrique, Théorie des types, Réutilisation.

1 INTRODUCTION

Le développement de langages fonctionnels typés (Standard ML [1], CAML [2]) et de langages de spécifications algébriques (CLEAR [3], PLUSS [4]) basés sur des théories formelles de types permet d'envisager un style de programmation dans lequel le type d'un programme serait donné à isomorphismes de types près. L'intérêt de programmer à isomorphismes de types près peut se faire ressentir lors de l'utilisation d'une librairie de programmes. En effet, si les types des programmes sont donnés à ismorphismes de types près, la manipulation de ces programmes est plus souple car l'adaptation systématique des interfaces des programmes, pour tenir par exemple compte de l'ordre des arguments, serait évitée. Dans [5, 6, 7], une autre motivation intéressante de la programmation à isomorphismes de types près est tirée des Bases de Données. Ici, il s'agit d'extraire une information, ou à défaut, toute information sémantiquement équivalente contenue dans la base. Formellement, le problème est d'extraire des informations à isomorphismes près dans une Base de Données.

2 ILLUSTRATION DU PROBLEME

La sémantique abstraite du type $string \times int \rightarrow int$ est la classe $[string \times int \rightarrow int]$, (où $[A]$ est la classe des types isomorphes à A), contient les types suivants:

1. $sting \times int \rightarrow int$ 2. $string \rightarrow int \rightarrow int$ 3. $string \times int \rightarrow int$ 4. $string \rightarrow int \rightarrow int$

Ces types sont obtenus en considérant les instances des isomorphismes naturels $A \times B \leftrightarrow B \times A$ (la commutativité notée *com* dans la suite) et $C^{A \times B} \leftrightarrow C^{B^A}$ (la currification, notée *cur*). Soit f une expression de type $string \times int \rightarrow int$, alors du point de vue de cette sémantique, toutes les expressions suivantes sont bien typées:

1. *f('ab', 2) : int* 2. f *'ab' : int→int* 3. f *(2, 'ab') : int* 4. f *2 : string→int*
avec *2 : int* et *'ab' : string* . Les exemples suivants montrent que tel n'est pas toujours le cas dans la pratique.

Exemple 2.1 Dans un langage fonctionnel typé, nous prenons ici le cas de CAML, si la fonction *f* est définie par la déclaration **fun** `f(s,n)=length(s)+n;;` dans laquelle *length : string→int* est une fonction prédéfinie qui donne la longueur d'une chaîne de caractères, le type attribué à *f* par CAML est *string*int→int* . Mais ici, seules les expressions 1) et 2) sont correctement typées. Donc la sémantique concrète donnée par CAML au type *string*int→int* contient uniquement les types *string*int→int* et *string→int→int* . Cette sémantique est donc plus restrictive que la sémantique abstraite qui prend en compte les deux isomorphismes *com* et *cur* alors que la sémantique concrète de CAML ignore l'isomorphisme de commutativité *com* .

Exemple 2.2 Soit *g* une fonction définie dans CAML par la déclaration **fun** `g(n,m)=2*n*m;;` de type *int*int→int* . Les types des arguments de g sont indistinguables; par conséquent, l'ordre des arguments est important dans l'évaluation de la fonction. En effet, une prise en compte de la commutativité *com* change le sens de la fonction car $g(com(n,m)) \neq g(n,m)$. Mais, l'exemple 2.1 suggère que lorsque les types des arguments ne sont pas identiques, *com* préserve le sens de la fonction; par conséquent, ne pas tenir compte de cet isomorphisme dans ce cas est une restriction sémantique.

Ces deux exemples montrent que le problème de la programmation à isomorphismes de types près est de déterminer les isomorphismes de types qui préservent le sens des programmes. Ces isomorphismes seront dits canoniques (section 4.3.4). L'exemple 2.2 suggère que $com: A \times B \leftrightarrow B \times A$ serait non canonique dans certains cas.

3 SPECIFICATION ET THEORIE DES TYPES

3.1 Théorie des types indépendants

La théorie des types indépendants la plus simple utilise uniquement les constructeurs $\times$, + et $\rightarrow$ ($A \rightarrow B$ est généralement noté B^A). Partant d'un certain nombre de types primitifs (*int, real,* ...), ces constructeurs permettent, grâce à des règles d'inférence, de définir tous les types permis dans le langage:

($\times$ intro) $\dfrac{A\ type \quad B\ type}{A \times B\ type}$ ($\rightarrow$ intro) $\dfrac{A\ type \quad B\ type}{A \rightarrow B\ type}$ (+ intro) $\dfrac{A\ type \quad B\ type}{A + B\ type}$

Dans une telle théorie, on peut interpréter certains types comme étant des formules de la logique intuitionniste [8]; les types primitifs sont les propositions de base et les constructeurs de

types correspondent aux connecteurs logiques. Ici, $\times$, + et $\rightarrow$ peuvent s'interpréter comme étant respectivement les connecteurs $\wedge$, $\vee$ et $\Rightarrow$. Un type est donc assimilable à une formule intuitionniste; une preuve de cette formule est un élément du type qui peut être assimilé à un programme. Des règles d'inférence permettent de dériver ces programmes:

$$\frac{a:A \quad b:B}{(a,b):A\times B} \qquad \frac{\begin{array}{c}[x:A]\\ \vdots \\ b:B\end{array}}{\lambda x.b : A\rightarrow B} \qquad \frac{a:A}{inl(a):A+B} \qquad \frac{b:A}{inr(b):A+B}$$

Une spécification n'ayant pas de solution est dite inconsistante. Dans une théorie des types indépendants, la plupart des spécifications consistantes admettent plusieurs solutions sémantiquement distinctes.

Exemple 3.1 Considérons la spécification *int* $\times$*string* , les programmes *snd* et *csta* définis par:

```
fun snd(x:int, y:string) = y ;;
fun csta(x:int, y:string) = 'ab' ,;
```

sont solutions de cette spécification. Ainsi donc, une spécification de la théorie ne détermine pas une solution particulière. Pour que cela soit le cas, il faut enrichir la théorie des types pour augmenter son expressivité.

3.1.1 Théorie des types dépendants

Une façon d'enrichir la théorie des types indépendants pour avoir des spécifications plus fines est d'introduire la notion de dépendance des types qui permet de construire des familles de types indexées par les éléments d'un autre type [9, 10]. La règle de formation des types dépendants est de la forme:

$$\frac{x:A}{B(x)\ Type}$$

qui se lit : $B(x)$ est un type pour tout élément x de A . Cette construction permet d'augmenter l'expressivité du langage par l'ajout de quantificateurs au moyen des règles de formation de produits et réunions disjointes généralisées:

$$\frac{\begin{array}{c}[x:A]\\ \vdots \\ B(x)\ Type\end{array}}{\Pi_{x:A}B(x)\ Type} \qquad \frac{\begin{array}{c}[x:A]\\ \vdots \\ B(x)\ Type\end{array}}{\Sigma_{x:A}B(x)\ Type}$$

dont la lecture est la suivant: si pour tout élément x de type A on a montré que $B(x)$ est un type, alors $\Sigma_{x:A}B(x)$ et $\Pi_{x:A}B(x)$ sont des types. Dans un langage muni de types dépendants, on peut

définir une fonction dont le type du résultat dépend du type des arguments. D'autre part, les théories de types dépendants offrent un cadre dans lequel les spécifications, les programmes et les relations d'implémentation peuvent être formalisées de manière uniforme. La notion de dérivation de programmes a été étudiée dans diverses théories de types [11, 12, 13]. Par contre, les divers aspects des Types de Données Abstraits n'ont pas encore été suffisament étudiés dans la théorie des types. Dans la section 3.1.3, nous montrons uniquement comment les types abstraits peuvent être spécifiés dan la théorie des types dépendants et grâce à ce lien, nous définissons, dans la section 4, une notion d'*équivalence canonique* des types abstrait au moyen des systèmes d'inférence de types.

3.1.2 Types et Propositions

En prenant des univers distincts TYPE et PROP pour les types et les propositions, on peut avoir une distinction conceptuelle entre les types de données manipulées par les programmes et les formules logiques exprimant les propriétés des programmes [14]. Les formules usuelles sont obtenues par les notations suivantes dans lesquelles $A \rightarrow B$ est écrit pour $\Pi_{x:A} B(x)$ lorsque A et B sont indépendants:

$$\begin{array}{lll}
\forall_{x:A} P(x) & =_{dl} & \Pi_{x:A} P(x) \\
\textbf{\textit{false}} & =_{dl} & \forall_{X:PROP} X \\
P \wedge Q & =_{dl} & \forall_{X:PROP} (P \rightarrow Q \rightarrow X) \rightarrow X \\
P \vee Q & =_{dl} & \forall_{X:PROP} (P \rightarrow X) \rightarrow (Q \rightarrow X) \rightarrow X \\
\neg P & =_{dl} & P \rightarrow \textbf{\textit{false}} \\
\exists_{x:A} P(x) & =_{dl} & \forall_{X:PROP} (\forall_{x:A} (P(x) \rightarrow X)) \rightarrow X
\end{array}$$

Des règles de formation permettent d'extraire des programmes à partir des preuves des formules. Une littérature abondante existe sur le sujet, on peut par exemple consulter [15, 16]. Nous donnons uniquement les règles de formation asociées au constructeur Σ qui nous permettra d'établir un lien entre les théories de types dépendants et les spécifications algébriques.

$$\frac{a:A \quad b:B[a/x]}{(a,b):\Sigma_{x:A} B} \qquad \frac{c:\Sigma_{x:A} B}{\pi_1(c):A} \qquad \frac{c:\Sigma_{x:A} B}{\pi_2(c):B[\pi_1(c)/x]}$$

3.1.3 Spécifications de types abstraits

L'opérateur Σ permet de spécifier des types abstraits algébriques. Pour simplifier les notations, nous adoptons la convention suivante:

$$\begin{bmatrix} x_1 : A_1 \\ \vdots \\ x_n : A_n \end{bmatrix} \quad =_{dl} \quad \Sigma_{x_1:A_1} \Sigma_{x_2:A_2} \cdots \Sigma_{x_{n-1}:A_{n-1}} A_n$$

et pour tout objet a de ce type $x_i[a]$ est la i^e composante de a . Lorsqu'il n'ya pas d'ambiguité sur l'objet a , $x_i[a]$ est tout simplement noté x_i.

Exemple 3.2 (Le type abstrait PILE d'ENTIERS) Soit le type *SORTE* défini par:

$$SORTE \quad =_{dl} \quad \begin{bmatrix} Dom : TYPE \\ eq : Dom \rightarrow Dom \rightarrow PROP \end{bmatrix}$$

Ce type peut servir à spécifier les sortes munies d'une relation binaire sur leur domaine. Une signature du type abstrait "Pile d'entiers naturels" est donnée par le Σ-type suivant:

$$Struc\_Pile \quad =_{dl} \quad \begin{bmatrix} pile : SORTE \\ vide : Dom[pile] \\ empiler : Nat \rightarrow Dom[pile] \rightarrow Dom[pile] \\ depiler : Dom[pile] \rightarrow Dom[pile] \\ sommet : Dom[pile] \rightarrow Nat \end{bmatrix}$$

où *Nat* est le type des entiers naturels. Le comportement attendu par un objet S de ce type est décrit par une formule *Ax_Pile(S)* dans la logique interne de la théorie (cf section 3.1.2):

$$\begin{array}{lll} Ax\text{-}Pile(S) \quad =_{dl} \quad Cong(eq) & \wedge & (A0) \\ \qquad eq(depiler(vide),\ vide) & \wedge & A1) \\ \qquad sommet(vide) =_{Nat} 0 & \wedge & (A2) \\ \qquad sommet(empiler(n,\ s)) =_{Nat} s & \wedge & (A3) \\ \qquad eq(depiler(empiler(n,\ s)),\ s) & & (A4) \end{array}$$

où l'axiome A0, *Cong(eq)* est une proposition exprimant que la relation binaire *eq* est une congruence et tous les axiomes ont universellement quantifiés sur *n:Nat* et *s:Dom[pile[S]]*. Notre type abstrait PILE D'ENTIERS NATURELS est donc défini par le Σ -type suivant:

$$PILE\_ENTIERS \quad =_{dl} \quad \begin{bmatrix} S : Struc\_Pile \\ p : Ax\_pile(S) \end{bmatrix}$$

$$=_{dl} \quad \Sigma_{s:struc\_Pile.\ p} : Struc\_Pile \rightarrow PROP$$

Un objet de ce type est une paire $(P,\ p)$ où P est une réalisation de la structure et p est une preuve que cette réalisation vérifie les axiomes; la détermination de la preuve p correspond à la phase de validation d'une implémentation.

Cet exemple nous permet de dire que dans la théorie des types dépendants, les types abstraits peuvent être définis comme étant les éléments du Σ-type suivant:

$$SPEC \quad =_{dl} \quad \begin{bmatrix} Struc : TYPE \\ Ax : Struc \rightarrow PROP \end{bmatrix}$$

4 EQUIVALENCE CANONIQUE

Dans cette section, nous montrons comment les systèmes d'inférence d'isomorphismes de types permettent de définir une équivalence canonique entre spécifications. L'équivalence canonique est la congruence minimale qu'on peut définir entre deux spécifications. Elle conerne la structuration des spécifications indépendamment de la façon dont elles sont construites.

4.1 Sémantique

En général, les sémantiques des théories de types et des spécifications algébriques sont exprimées dans la théories des catégories [4, 17] ou plus généralement dans des institutions [18]. Mais, les objets d'une catégorie sont toujours considérés à isomorphismes près; Par conséquent, deux types (ou deux spécifications) isomorphes sont modélisés par le même objet de la catégorie. Mais dans la plupart des languages de spécification, la sémantique est étroitement liée au formalisme de représentation syntaxique. C'est ainsi que dans la plupart des langages, deux spécifications ayant des signatures différentes sont distinctes même si elles ont le même comprtement (c'est à dire les mêmes modèles). Dans [19], une illustration de ce problème est donnée pour le langage de spécification PLUSS dont la sémantique est révisée pour tenir compte d'une opération purement syntaxique: le renommage des variables. Une question vient donc à l'esprit:

> Supposons que deux spécifications admettent les mêmes modèles dans une certaine structure de catégories, existe-t-il un moyen de détecter à priori que les deux spécifications sont isomorphes ?

Comme moyen de détection des isomorphismes de spécifications, nous introduisons la notion de *système d'inférence d'isomorphismes.*

4.2 Systèmes d'inférence d'isomorphismes

Définition 4.1 Un système d'inférence d'isomorphismes dans une structure de catégories $\boldsymbol{CAT}$ est une paire $<\boldsymbol{A}, \boldsymbol{R}>$ où :

$\boldsymbol{A}$ est un ensemble fini de couples de morphismes $(t:A\rightarrow B,\ t':B\rightarrow A)$ tels que $t;t' = idA$ et $t';t = idB$. Le couple (t, t') est appelé isomorphisme de base (ou axiome) et est dénoté indifféremment par un de ses membres en fonction du sens prioritaire qu'on voudrait donner à l'isomorphisme en cas de réécriture.

$\boldsymbol{R}$ est un ensemble fini de règles d'inférences comprenant au moins les deux règles suivantes:

[SUBST] Règle d'instanciation : pour toute substitution ω, on a :

$$\frac{\vdash t:A\rightarrow B}{\vdash \omega(t):A\rightarrow B}$$

[COMP] Règle de composition:

$$\frac{\vdash t : A \to B \quad \vdash r : B \to C}{\vdash t;r : A \to C}$$

où $\vdash$ est le symbole de dérivation associé au système d'inférence.

Dans la suite, $\mathbf{CAT}(A)$ dénote la classe des modèles de A dans la structure de catégories $\mathbf{CAT}$ et $A \cong_t B$ est écrit pour $\vdash t : A \to B$ et $A \cong B$ est écrit pour "il existe t tel $\vdash t : A \to B$". Remarquons que si $A \cong_t B$ alors l'isomorphisme t est un couple de programmes qui permet d'implémenter A dans B et réciproquement.

4.3 Propriétés des systèmes d'isomorphismes

Un bon système d'isomorphismes de types $\mathbf{H}$ dans un domaine sémantique $\mathbf{CAT}$ doit avoir les propriétés suivantes :

(a) Adéquation : $\mathbf{H} \vdash A \cong B$ implique $\mathbf{CAT}(A) = \mathbf{CAT}(B)$.

(b) Cohérence: $\mathbf{H} \vdash A \cong_t B$ et $\mathbf{H} \vdash A \cong_{t'} B$ implique $t = t'$. Cette propriété garantit que la relation $\cong$ est une égalité: si $\cong B$ alors il ya une façon unique d'implémenter A dans B et réciproquement en utilisant las isomorphismes du système.

(c) Complétude : $\mathbf{CAT}(A) = \mathbf{CAT}(B)$. implique $\mathbf{H} \vdash A \cong B$. La complétude garantit que le système d'iomorphimes détermine tous les types équivalents. En général, on peut se contenter d'une complétude faible sur un sous domaine $\mathbf{CAT}'$ de $\mathbf{CAT}$: $\mathbf{CAT}'(A) = \mathbf{CAT}'(B)$ implique $\mathbf{H} \vdash A \cong B$.

(d) Décidabilité : $\mathbf{H}$ est décidable s'il existe un algorithme de décision de $\mathbf{H} \vdash A \cong B$. Un tel algorithme de décision construit les isomorphismes dans les cas positifs. Remarquons qu'un sytème d'inférence d'isomorphismes est un système équationnel dont la décidabilité peut être étudiée à l'aide des techniques standard de réécriture [20].

4.4 Isomorphismes canoniques

Soient $\mathbf{H}$ et $\mathbf{H}'$ deux systèmes d'isomorphismes dans $\mathbf{CAT}$, alors $A \cong^{\mathbf{H}} B$ ssi $A \cong^{\mathbf{H}'} B$ mais on peut très bien avoir $\mathbf{H} \vdash t : A \to B$ et $\mathbf{H}' \vdash t' : A \to B$ avec $t \neq t'$. Autrement dit, $\mathbf{H}$ et $\mathbf{H}'$ déterminent les mêmes classes d'équivalences mais peuvent définir des conversions différentes entre types équivalents. Par conséquent, un système d'isomorphismes est un système de construction de morphismes d'implémentation entre types équivalents. Chaque système d'isomorphismes donne une sémantique particulière à l'équivalence des types dans la mesure où définir un tel système revient à fixer les conversions de types considérées comme étant canoniques, c'est à dire naturelles; tout autre isomorphisme ne pouvant être construit par le

système est non canonique en ce sens qu'il ne respecte pas la sémantique donnée à l'équivalence par ce système d'isomorphismes. D'où la définition suivante:

Définition 4.2 Soit $\mathbf{H}$ un système d'isomorphismes, deux types (ou spécifications) Sp et Sp' sont canoniquement isomorphes (équivalents) dans $\mathbf{H}$ si on a $Sp \cong^{\mathbf{H}} Sp'$.

5 EXEMPLE DE SYSTEME

Nous considérons ici la structure des Catégories Cartésiennes Closes (CCC) qui donne la sémantique de la plupart des théories de types dépendants. Dans [21], une caractérisation des types isomorphes dans toutes les CCC est donnée:

Fait 5.1 Deux types sont isomorphes dans CCC ssi ils sont équivalents dans la Logique Intuitioniste Propositionelle (IPC) qui est la logique interne de la théorie des types indépendants.

Un sytème d'inférence d'isomorphismes cohérent, complet et décidable établi dans [22] (partant de conidérations différentes) est le suivant:

5.1 Les axiomes

1. $com(A, B) : A \times B \leftrightarrow B \times A$
2. $ass(A, B, C) : (A \times B) \times C \leftrightarrow A \times (B \times C)$
3. $idl(A) : A \times 1 \leftrightarrow A$
4. $expl(A) : A^1 \leftrightarrow A$
5. $cur(A, B, C) : C^{A \times B} \leftrightarrow C^{B^A}$
6. $exp(A) : 1^A \leftrightarrow 1$
7. $twist(A, B, C) : (B \times C)^A \leftrightarrow B^A \times C^A$
8. $id(A) : A \leftrightarrow A$

Les morphismes de conversion, assez évidents sont omis par souci de clarté. Dans ces axiomes, les types A, B et C sont syntaxiquement distincts; c'est à dire que l'axiome 2 par exemple ne doit pas être instancié en $(A \times A) \times A \leftrightarrow A \times (A \times A)$. Cette constrainte syntaxique élimine des cas trivialement non canoniques comme celui présenté à l'exemple 2.2 (qui se réfère à l'axiome 1). Dans un contexte où les déclarations de noms sont admises, cette contrainte permet de définir une équivalence par nom.

5.2 Les règles d'inférence

En plus des règles *SUBST* et *COMP*, on a les deux règles suivantes:

$$[\text{PROD}] \quad \frac{u : A \to B \quad v : C \to D}{u \otimes v : A \times C \to B \times D}$$

$$[\text{EXP}] \quad \frac{u : B \to A \quad v : C \to D}{v^u : C^A \to D^B}$$

avec $u \otimes v(<x, y>) = < u(x), v(x) >$ et $v^u(f) = \lambda y{:}B.\, v(f(u(y)))$.

Exemple 5.1 Il est facile de dériver que $(id_{C \times E})^{uss(A,B,C)} ; twist(A \times (B \times C), C, E)$ est un ismorphisme canonique entre $(C \times E)^{(A \times B) \times C}$ et $C^{A \times (B \times C)} \times E^{A \times (B \times C)}$.

L'algorithme de décidabilité de cette famille d'isomorphismes est amélioré dans [6] pour des raisons d'efficacité de l'implémentation. Une extension de ce résultat aux théories de types dépendants est abordée dans [23]. Dans [24], la détection des isomorphismes de spécifications modulaires est étudiée dans une catégorie de diagrammes finiment cocomplète. Un système d'inférence d'isomorphimes (dont les propriétés ne sont pas étudiées) est sous-jacente à la méthode développée.

6 CONCLUSION

Dans ce papier, nous avons établi un cadre théorique permettant d'étudier l'équivalence des spécifications dans divers domaines sémantiques. Les systèmes d'inférence d'isomorphismes permettent de définir des équivalences de manière constructive. En effet, étant donné une librairie de spécifications de base (qui forme une catégorie $\mathcal{C}_0$) et un ensemble de directives d'assemblage (limites, colimites, pullbacks, pushouts, etc ...) qui permettent de combiner des spécifications de façon modulaires, un système d'inférence d'isomorphismes permet de construire des morphismes de conversion entre spécifications équivalentes à partir des isomorphismes entre spécifications de base en utilisant les règles d'inférence du système qui doivent par conséquent préserver l'équivalence (adéquation, cohérence et complétude). Plus généralement, un système d'inférence de morphismes distingués (isomorphismes, monomorphismes, etc ...) permet d'envisager une construction de morphismes de réutilistion de spécifications (ou de programmes).

Références

[1] Milner R., *A proposal for Standard ML* , in: Proc. ACM Symposium on Lisp and Functional Programming, Austin Texas (1984)

[2] Cousineau G., *The Categorical Abstract Machine* , in: (Ed. G. Huet) Logical Foundations of Functional Programming, pp 25-40, Addison Wesley NY (1990)

[3] Burstall R., Goguen J., *The semantics of Clear, a specification language . Abstract Software Specifications*, in: (Ed. D. Bjorner) Proc. 1979 Copenhagen Winter School, LNCS 86, pp 293-332 Springer Verlag Berlin (1980)

[4] Bidoit M., *Pluss, un langage pour le développpement de spécifications algébriques modulaies*, Thèse d'Etat, Université Paris-Sud (1989)

[5] Rittri M., *Using types as keys in functional libraries*, in: Proc. 4th international conference of functional programming languaues and computing architectures, ACM Addison Wesley (1989)

[6] Rittri M., *Retrieving library identifiers of types at least as general, modulo CCC-isomorphims, as a given type*, Dpt CS, Chalmers Univ. Tech., PMG Report S-412 96

Göteborg (1990)
[7] Rittri M., *Retrieving library functions by unifying types modulo linear isomorphisms*, Dpt CS, Chalmers Univ. Tech., PMG Report 66 Göteborg (1992)
[8] Coquand T., *Sur l'analogie entre Propositions et Types*, in: (Ed. G. Cousineau, P.L. Curien, B. Robinet) Combinators and Functional Programming Languages, LNCS 242, pp 87-126, (1986)
[9] Girard J.Y., *The System F of variables, Fifteen years later*, Theoretical Computer Science 45, pp 159-192 (1980)
[10] Martin-Löf P., *An Intuitionistic Theory of Types: Predicative Part*, in: (Ed. H. Rose, J. Stephen) Logic Colloquium 73, pp 73-118, North-Holland Amsterdam (1974)
[11] Nordström B., Peterson K. Smith, *Programming in Martin Löf's Type Theory: an introduction*, Oxford University Press (1990)
[12] Backhouse R., Chistolm P., Malcom G., *Do-it-yourself Type Theory*, Formal Aspects of Computing 1(1) (1989)
[13] Paulin-Mohring C., *Extracting F^{ω} programs from proofs in the Calculus of Constructions*, Proc. 6^{th} annual ACM symposium on Principles of Programming Languages POPL'89 (Austin, Texas) January (1989)
[14] Zhaolui Luo, *An Extended Calculus of Constructions*, PhD Thesis Univ. Edinburg (1990)
[15] Burstall R., Mackinsa J., *Deliverables: an approach to program development in the Calculus of Constructions*, Proc. 1rst Workshop on Logical Frameworks (1990)
[16] Martin-Löf P., *Constructive mathematics and computer science*, in: Logic, Methodology and Philosophy of Sciences VI, pp 153-175 (1980)
[17] Hyland J. M., Pitts A., *The Theory of Construction: categorical semantics and set theoretic models*, Contemparary Mathematics, Vol 92 (1989)
[18] Goguen J., Burstall R., *Institutions: Abstract Model Theory for specification and programming*, Journal of ACM 39(1) pp 95-146 (1992)
[19] Roques C., *Modularité dans les spécifications algébriques: Théorie et applicatiuons*, Thèse de Doctorat, Université Paris-Sud (1994)
[20] Knuth D.E., Bendix P.B., *A simple Word Problem in Universal Algebras*, in: (Ed. J. Leech) Computational Problems in Abstract Algebra, Pergamon Press Oxford (1969)
[21] Bruce K. and al., *Provable isomorphisms of types types*, Rapport de recherche Laboratoire informatique ENS Paris (1990)
[22] Soloviev S. V., *The Category of finite sets and Cartesian Closed Categories*, Journal of Soviet Mathematics 22(3) (1983)
[23] Fouda Ndjodo M., *Systèmes de réécriture et cohérence des isomorphisme de types dans les catégories localement closes*, Thèse de Doctorat Univ. Aix-Marseille II (1992)
[24] Oriat C., *Detecting Isomorphisms of Modular Specifications with Diagrams*, Proc. of AMAST'95, Montréal, July 1995, LNCS 936, pp 184-198 (1995)