Cours 2-5 : Démonstration Automatique
Cours avancé : combinaison de procédures de décision

Claude Marché
mailto:Claude.Marche@lri.fr

18 décembre 2007

Table des matières

1 Motivations

1.1 Exemple d'une preuve de programme

Le petit programme Java suivant est annoté en JML (Java Modeling Language, `http://www.jmlspecs.org`)

```
class Purse {
    int balance;
    //@ public invariant balance >= 0;

    /*@ normal_behavior
      @  requires s >= 0;
      @  ensures balance == \old(balance) + s;
      @*/
    public void credit(int s) {
        balance += s;
   }
}
```

Un outil comme Krakatoa (`http://krakatoa.lri.fr`) est capable de générer des *obligations de preuve*, qui sont des formules logiques du premier ordre dont la validité implique la correction du programme vis-à-vis de ses spécifications. Sur cet exemple très simple, une obligation de preuve est générée :

$$\begin{array}{l}
\forall\, \mathit{this},\ s,\ \mathit{balance},\ s \geq 0 \land \mathtt{Purse_invariant}(\mathit{balance},\mathit{this}) \rightarrow \\
\quad \forall\, \mathit{tmp},\ \mathit{tmp} = \mathit{select}(\mathit{balance},\mathit{this}) + s \rightarrow \\
\quad\quad \forall\, \mathit{balance}_0,\ \mathit{balance}_0 = \mathit{store}(\mathit{balance},\mathit{this},\mathit{tmp}) \rightarrow \\
\quad\quad\quad \mathit{select}(\mathit{balance}_0,\mathit{this}) = \mathit{select}(\mathit{balance},\mathit{this}) + s \land \mathtt{Purse_invariant}(\mathit{balance}_0,\mathit{this})
\end{array}$$

où

$$\mathtt{Purse\_invariant}(\mathit{balance},\mathit{this}) := \mathit{select}(\mathit{balance},\mathit{this}) \geq 0$$

Cette formule est valide non pas en général mais quand $\geq$ est interprété dans $\mathbb{Z}$, et quand *select* et *store* sont interprétés dans la *théorie des tableaux* :

$$\begin{array}{rcl}
\forall\, m,\ x,\ v,\ \mathit{select}(\mathit{store}(m,x,v),x) & = & v \\
\forall\, m,\ x,\ y,\ v,\ x \neq y \rightarrow \mathit{select}(\mathit{store}(m,x,v),y) & = & \mathit{select}(m,y)
\end{array}$$

Cette formule peut être prouvée interactivement (par exemple avec Coq, `http://coq.inria.fr`), mais aussi automatiquement (par exemple avec Simplify, `http://research.compaq.com/SRC/esc/simplify`).

1.2 Besoins

On a donc besoin de pouvoir décider la validité de formules logiques :
- du premier ordre arbitraire : quantificateurs $\forall$ et $\exists$ avec alternation et imbrication quelconque ;
- avec un prédicat d'égalité
- avec des symboles de prédicats et de fonctions interprétées dans des structures particulières :
  - arithmétique linéaire sur N, Z, Q, R, ou plus rarement non linéaire (cad avec multiplication)
  - théories de structures de données : tableaux ou listes, etc.

Si les structures considérées sont finiment axiomatisables en logique du premier ordre, on peut se ramener au cas de la décision de la validité en logique du premier ordre pure : ϕ est valide pour des structures axiomatisées par T ssi $T \rightarrow \phi$ est valide en général. Problèmes de cette approche :

1. certaines structures ne sont pas finiment axiomatisable : par exemple l'arithmétique linéaire.
2. même pour une structure finiment axiomatisable, il peut être bien plus efficace de raisonner dessus avec un mécanisme spécifique qu'avec des mécanismes généraux : procédures de décisions adaptées à ces structures.

1.3 Historique

La difficultée posée alors est de combiner des procédures de raisonnement général de logique du premier ordre, et des procédures de décision spécifiques. Historiquement, dans les années 1980, deux *schémas de combinaison* de procédures ont été proposés indépendemment par Nelson et Oppen [5, 6] et par Shostak [10, 11]. Ces deux schémas permettent de construire une procédure de décision pour l'union de deux structures, lorsque celles-ci sont *disjointes* c'est-a-dire ne partagent pas de symbole de fonction ou de prédicat (excepté l'égalité).

De nombreux travaux ont poursuivis les deux précédents. Ce cours se base en grande partie sur un article récent de Conchon et Krstic [2] qui propose une présentation unifiée des méthodes précédentes.

2 Problématique de la combinaison de théories

On considère que le seul prédicat est l'égalité. Les signatures considérées ici ne concernent donc que des symboles de fonctions. Une Σ-théorie est une ensemble de formules du premier ordre construite sur le prédicat d'égalité et les symboles de fonction de Σ.

La motivation ci-dessus s'intéresse à prouver la validité d'une formule. Pour le raisonnement automatique, il est habituellement plus simple de raisonner par réfutation : montrer que la conjonction des hypothèses et de la négation du but est insatisfaisable. Dans la suite on va donc s'intéresser plutôt à la décision de la satisfaisabilité que de la validité.

2.1 Union disjointe de théories

On suppose fixées deux signatures Σ_1 et Σ_2 *disjointes* (c'est-à-dire $\Sigma_1 \cap \Sigma_2 = \emptyset$) et l'on pose $\Sigma = \Sigma_1 \cup \Sigma_2$. On fixe deux théories T_1 et T_2 sur Σ_1 et Σ_2 respectivement. $T = T_1 \cup T_2$ est alors une Σ-théorie appelée *union disjointe* de T_1 et T_2. On généralise naturellement cela à l'union disjointe d'un nombre fini de théories $T_1, \ldots, T_n$.

On dit qu'un terme est i-pur si tous ses symboles sont dans Σ_i.

On cherche à construire une procédure de décision de la satisfiabilité dans T, étant données des procédures de décision pour T_1 et T_2.

On va traiter le problème en 3 étapes : d'abord le cas des conjonctions closes, puis le cas des formules closes quelconques, et enfin le cas général.

3 Cas des conjonctions closes

Deux travaux marquants ont été réalisés dans les années 1980. Nelson et Oppen [5] ont proposé (et prouvé) un algorithme de combinaison assez général, puis Shostak [11] a proposé un algorithme plus efficace s'appliquant sur un cas moins général. Il est remarquable que la correction de la méthode de Shostak est restée obscure très longtemps, des preuves de corrections (de variantes) ayant été faites seulement en 2002 [1, 9, 4]. On peut aussi mentionner une preuve récente de correction de Nelson-Oppen, faite en 1996 [12], plus abstraite que la preuve originale, qui sera utilisée dans ce cours.

3.1 Méthode abstraite de propagation d'égalités

Idée principale : abstraction des termes non purs, pour transformer la conjonction donnée en un ensemble de conjonctions de littéraux purs. Puis transmettre les conjonctions i-pures à la procédure de décision pour T_i.

Mais attention, ce n'est pas complet ainsi, comme le montre l'exemple de l'obligation de preuve précédente, explicité en annexe. L'idée de Nelson et Oppen est de demander aux procédures de décision de répondre, quand c'est satisfaisable, une disjonction d'égalités impliquées (voir annexe).

Définition 3.1 *Une* configuration propre *est un triplet* $\Delta; \Gamma; \Phi_0, \ldots, \Phi_n$ *où*

- Δ *: ensemble de disjonctions de littéraux* simples*, i.e.* $x \bowtie y$ *où* x *et* y *sont des variables et* $\bowtie$ *est* $=$ *ou* $\neq$.
- Γ *: ensemble de littéraux sur* Σ.
- Φ_i *: ensemble d'équations de la forme* $x = t$ *où* t *est un terme* i*-pur.*

Une configuration *est une configuration propre ou bien* $\perp$.

Une configuration est dite satisfaisable si elle est propre et $\Delta \wedge \Gamma \wedge \Phi_0 \wedge \cdots \wedge \Phi_n$ *est satisfaisable dans* T.

Une procédure de décision par propagation d'égalité est un algorithme qui prend en entrée une conjonction Γ et qui produit une séquence de configurations à partir de la configuration initiale $\emptyset; \Gamma; \emptyset$, utilisant les règles de transformation de la figure 1. La règle Contradict suppose que l'on

Abstract

$$\frac{\Delta;\Gamma \cup \{A[s]_p\};\dots,\Phi_i,\dots}{\Delta;\Gamma \cup \{A[z]_p\};\dots,\Phi_i \cup \{z = s\},\dots}$$

si s est un terme i-pur non variable, et z variable fraîche.

Arrange

$$\frac{\Delta;\Gamma \cup \{x \bowtie y\};\Phi_0,\dots,\Phi_n}{\Delta \cup \{x \bowtie y\};\Gamma;\Phi_0,\dots,\Phi_n}$$

si x et y sont des variables.

Deduct

$$\frac{\Delta;\Gamma;\Phi_0,\dots,\Phi_n}{\Delta \cup \delta;\Gamma;\Phi_0,\dots,\Phi_n}$$

si $\Lambda \subseteq \Delta$ est une *requête* (une conjonction d'égalités entre variables), et δ est une réponse pertinente cad une disjonction d'égalités entre variables telle que $\Phi_i, \Lambda \models_{T_i} \Lambda \rightarrow \delta$ et $\Delta \not\models \delta$.

Contradict

$$\frac{\Delta;\Gamma;\Phi_0,\dots,\Phi_n}{\bot}$$

si $\Phi_i \wedge \Delta$ insatisfaisable.

Branch

$$\frac{\Delta \cup \{x_1 = y_1,\dots,x_k = y_k\};\Gamma;\Phi_0,\dots,\Phi_n}{\Delta \cup \{x_i = y_i\};\Gamma;\Phi_0,\dots,\Phi_n}$$

si $\Delta \not\models x_i = y_i$ pour un i, $1 \leq i \leq k$.

FIG. 1 – Règles de la propagation d'égalités abstraite

possède une procédure de décision pour chaque théorie T_i. La règle Deduct suppose de plus que l'on possède une procédure pour chaque T_i qui est capable de produire toute disjonction impliquée par Φ_i et une requête Λ. Enfin, la règle Branch fournit un raisonnement par cas pour traiter les disjonctions d'égalités.

Lemme 3.2 *La relation $\Rightarrow$ termine*

Preuve. On mesure la complexité des configurations par
- la taille de Γ, cad la somme des tailles de ses éléments.
- l'ensemble Δ, ordonné par l'ordre d'implication : $\Delta \succ \Delta'$ ssi $\Delta' \models \Delta$ et $\Delta \not\models \Delta'$.

Alors les règles Abstract et Arrange font décroire la première composante, alors que Deduct et Branch laissent Γ constant, ainsi que l'ensemble des variables de la configuration, et font décroitre Δ. On conclut en remarquant que pour un ensemble fixé de variables, l'ordre d'implication est bien fondé (car il n'y a qu'un nombre fini de Δ possibles !).

Enfin, la règle Contradict ne pose pas de problème puisqu'elle termine toujours une réduction.

Définition 3.3 *Un algorithme de décision par propagation applique les règles de la figure 1, et cherche à dériver une configuration propre irréductible. (Ceci signifie qu'en cas de branchement, si l'on aboutit à $\perp$ il faut backtracker.)*

3.2 Preuve de correction et complétude

L'algorithme ainsi obtenu n'est pas complet en général, il faut supposer que toutes les théories T_i sont stables-infinies. On utilise alors le théorème 3.5 suivant, sur les modèles.

Définition 3.4 *Une théorie T est stable-infinie si toute formule satisfiable admet un modèle infini.*

Exemple de théorie non stable-infinie : théorie d'un type fini, par exemple les booléens.

Proposition 3.5 (Tinelli & Harandi [12]) *Considérons des théories $T_1 \ldots, T_n$ stables-infinies, et des formules Φ_i. Soit $\vec{v}$ est l'ensemble des variables de ces formules, et $D(\vec{v})$ la formule $\bigwedge_{x,y\in\vec{v},x\neq y} x \neq y$. Si pour tout i $\Phi_i \cup D(\vec{v})$ est satisfaisable, alors $\Phi_0 \wedge \cdots \wedge \Phi_n$ est satisfaisable.*

Lemme 3.6 *Toute configuration propre irréductible est satisfaisable.*

Preuve. cf [2]

Lemme 3.7 *Si $C \Rightarrow C'$ par une règle autre que Branch, alors C et C' sont équi-satisfaisables.*

Preuve. cf [2]

Lemme 3.8 *Si $C \Rightarrow C'$ par Branch, alors*
- *Si C' est satisfaisable, alors C' est satisfaisable.*
- *si C est satisfaisable, alors il existe une réduction par Branch $C \rightarrow C''$ telle que C'' est satisfaisable.*

Preuve. cf [2]

Théorème 3.9 *Un ensemble de formules* Γ *est satisfiable ssi il existe une configuration* C *propre irréductible telle que* $\emptyset;\Gamma;\emptyset \Rightarrow^* C$.

Preuve. Il suffit de montrer que pour toute configuration C, C est satisfiable ssi il existe C' propre irréductible telle que $C \Rightarrow^* C'$. On raisonne par induction par rapport à $\Rightarrow$, qui est bien-fondée grâce au lemme 3.2.

Si C est irréductible, le lemme 3.6 permet de conclure. Si C est réductible en C', si c'est par une règle autre que Branch, on utilise le lemme 3.7 et on conclut par l'hypothèse de récurrence sur c', et si c'est par Branch on utilise le lemme 3.8 et on applique l'hypothèse de récurrence sur C' et C''.

3.3 Instance Nelson-Oppen

L'algorithme de Nelson-Oppen original est une instance simple de l'algorithme précédent. Il s'applique à des théories (toujours stables-infinies) dites *convexes*.

Définition 3.10 *Une théorie* T *est* convexe *si pour toute conjonction* Λ *de littéraux et tous termes* $a_1, b_1, \ldots, a_k, b_k$, *si* $T \models \Lambda \rightarrow a_1 = b_1 \vee \cdots \vee a_k = b_k$ *alors il existe* i *tel que* $T \models \Lambda \rightarrow a_i = b_i$.

Par exemple, toutes les théories axiomatisées par des clauses de Horn sont convexes. Malheureusement c'est à peu près tout... ainsi l'arithmétique linéaire sur $\mathbb{Z}$, ou les tableaux, ne sont pas des théories convexes.

Dans le cas de théories convexes, une procédure de décision renverra des réponses qui seront des égalités simples, au lieu de disjonctions. Par conséquent, aucune disjonctions n'est introduite, et la règle Branch n'est plus nécessaire.

L'algorithme original de Nelson-Oppen correspond alors à la stratégie

$$Abstract*.Arrange*.(Contradict|Deduct)*$$

3.4 Instance Shostak

On a supposé que pour chaque théorie T_i, non seulement on avait une procédure de décision qui sait dire si Φ_i est insatisfiable dans T_i, mais aussi que pour toute requête Λ, on doit savoir déduire toute disjonction δ qui est conséquence de Φ_i et Λ. En principe, cette deuxième procédure peut être construire à partir de la première : il suffit d'énumérer les δ possibles (il y en a un nombre fini car il y a un nombre fini de variables en jeu), et de tester si $\Phi_i \wedge \Lambda \wedge \neg\delta$ est insatisfiable. C'est évidemment particulièrement inefficace.

L'idée de Shostak correspond à proposer un schéma général plus efficace pour une telle procédure de production de réponse à une requête.

Définition 3.11 *Un* canoniseur *pour une théorie* T *est une fonction canon des termes clos dans les termes clos, telle que*

– $T \models s = t$ *ssi* $canon(s) = canon(t)$.
– $canon(canon(s)) = canon(s)$
– $Var(canon(s)) \subseteq Var(s)$
– *si* $canon(s) = s$ *alors* $canon(t) = t$ *pour tout sous-terme* t *de* s.

On dit que $canon(s)$ *est le représentant canonique de* s.

Un solveur *pour une théorie* T *est une fonction solve qui à toute équation* $s = t$ *associe* $\perp$ *si elle n'est pas satisfaisable dans* T*, ou bien si elle est satisfaisable, retourne un unificateur plus général sous la forme d'une conjonction* $x_1 = t_1, \ldots, x_n = t_n$.

Une théorie de Shostak *est une théorie convexe qui possède un canoniseur et un solveur.*

Pour une théorie T_i de Shostak, on remplace la règle Deduct par les règles de la figure 2. La fonction D donne pour chaque variable de Δ un représentant canonique pour la relation d'équivalence $\Delta \models x = y$, et est étendue à tous les termes par $D(f(t_1, \ldots, t_n)) = f(D(t_1), \ldots, D(t_n))$.

4 Cas des propositions closes

Difficulté : traiter les disjonctions et les négations (bien que : Shostak supporte les négations...)

Principe : combiner avec une procédure de décision de SAT (problème de satisfiabilité en logique propositionnelle). Par exemple BDD, principe *branch-and-bound*, méthode de Davis et Putnam.

Méthode naïve : mettre en DNF, interroger la procédure de décision sur chaque conjonction. Peut facilement exploser pour 2 raisons : le calcul de la DNF peut être très long, et même si la DNF est calculée en temps raisonnable, elle peut contenir un grand nombre de conjonctions et donc il est très lourd de les énumérer.

Pour éviter de calculer la DNF : principe *branch-and-bound*, qui revient à calculer des conjonctions « à la volée ». Les BDD ont un effet analogue.

Pour éviter d'énumérer les conjonctions [1] : interroger la procédure de décision avec une première conjonction, si satisfiable on a fini, sinon, la procédure de décision peut renvoyer un sous-ensemble de littéraux non satisfiable, on peut alors recommencer le test de SAT en ajoutant la négation de cette conjonction de littéraux.

Il existe aussi une approche intégrée : ICS [3] (destinée à être utilisée par l'assistant de preuve PVS, `http://pvs.csl.sri.com`), qui est donc plus complexe, mais laisse espérer plus d'efficacité. Cette méthode utilise l'idée des BDD pour étendre l'approche Shostak aux formules propositionnelles quelconques au lieu de conjonctions seulement.

5 Cas des formules arbitraires

2 écoles :

– procédure de sélection d'instances closes par des heuristiques. C'est incomplet, mais en pratique ca marche bien (Simplify).
– Utiliser l'unification, cad un raisonnement par paramodulation (haRVey [7, 8]).

Cela reste de nos jours trop peu étudié, c'est encore un sujet de recherche très actif !

TDeduct

$$\frac{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=s,y=s\},\ldots}{\Delta\cup\{x=y\};\Gamma;\ldots,\Phi_i\cup\{x=s,y=s\},\ldots}$$

si $D(x)\neq D(y)$

Subst

$$\frac{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=s\},\ldots}{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=D(s)\},\ldots}$$

si $s\neq D(s)$.

Canonize

$$\frac{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=s\},\ldots}{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=canon_i(s)\},\ldots}$$

si $s\neq canon_i(s)$

Solve

$$\frac{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=s,y=t\},\ldots}{\Delta;\Gamma;\ldots,\Phi_i\cup\{x=s,y=t,solve(s=t)\},\ldots}$$

si $D(x)=D(y)$, $s\neq t$, $s=t$ T_i-satisfaisable

FIG. 2 – Règles de déduction avec canoniseur et solveur

A Exemple complet de la preuve de l'obligation issue du programme Java simple

On veut montrer la validité de

```
forall this, s, balance,
  s >= 0 /\ Purse_invariant(balance,this) ->
  forall tmp,
    tmp = select(balance,this) + s ->
    forall balance0,
      balance0 = store(balance,this,tmp) ->
      select(balance0,this) = select(balance,this) +  s /\
```

```
      Purse_invariant(balance0,this)
```

avec

```
Purse_invariant(balance,this) := select(balance,this) >= 0
```

Négation du but + dépliage de `Purse_invariant`

On se ramène à montrer l'insatisfaisabilité de

```
s >= 0 /\
select(balance,this) >= 0 /\
tmp = select(balance,this) + s /\
balance0 = store(balance,this,tmp) /\
not (select(balance0,this) = select(balance,this) +  s /\
      select(balance0,this) >= 0)
```

La mise en forme normale disjonctive donne 2 conjonctions :

```
s >= 0 /\
select(balance,this) >= 0 /\
tmp = select(balance,this) + s /\
balance0 = store(balance,this,tmp) /\
select(balance0,this) <> select(balance,this) +  s
```

et

```
s >= 0 /\
select(balance,this) >= 0 /\
tmp = select(balance,this) + s /\
balance0 = store(balance,this,tmp) /\
select(balance0,this) < 0
```

On se concentre sur la deuxième conjonction : il n'est pas difficile de voir que cette conjonction est satisfaisable dans l'arithmétique, et aussi satisfaisable dans la théorie des tableaux, c'est dans l'union des deux théorie qu'elle ne l'est pas.

$\rightarrow$ Comment combiner une procédure de décision pour l'arithmétique et une procédure de décision pour les tableaux ?

Abstraction

Partie arithmétique :

```
s >= 0 /\
x >= 0 /\
tmp = x + s /\
y < 0
```

Partie tableaux :

```
x = select(balance,this) /\
balance0 = store(balance,this,tmp) /\
y = select(balance0,this)
```

Déduction

Si on passe la partie 2 à une procédure de décision sur les tableaux, celle-ci peut déduire des choses :

```
x = select(balance,this) /\
balance0 = store(balance,this,tmp) /\
y = select(balance0,this)
```

implique

```
y = select(store(balance,this,tmp),this) = tmp
```

→ une nouvelle égalité entre variables

Propagation

l'égalité `y = tmp` est propagée, et cette fois la procédure de décision d'arithmétique est capable de déduire l'insatisfaisabilité de

```
s >= 0 /\
x >= 0 /\
tmp = x + s /\
y < 0 /\
y = tmp
```

C'est gagné !

Références

[1] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249, Copenhagen, Denmark, 2002. Springer.

[2] S. Conchon and S. Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354(2) :187–210, 2006. Special Issue of TCS dedicated to a refereed selection of papers presented at TACAS'03.

[3] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS : Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2001.

[4] H. Ganzinger. Shostak light. In *Proceedings of the 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 332–347, Copenhagen, Denmark, 2002. Springer.

[5] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2) :245–257, Oct. 1979.

[6] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27 :356–364, 1980.

[7] S. Ranise and D. Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, Sept. 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[8] S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the extended canonizer : A family picture with a newborn. In *First International Colloquium on Theoretical Aspects of Computing*, Lecture Notes in Computer Science, Guiyang, China, Sept. 2004. Springer.

[9] N. Shankar and H. Rueß. Combining Shostak theories. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2378 of *LNCS*, pages 1–19, Copenhagen, Denmark, 2002. Springer.

[10] R. Shostak. Deciding combinations of theories. Technical Report CSL 132, SRI International, Feb. 1982.

[11] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31 :1–12, 1984.

[12] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Proceedings of the First International Workshop on Frontiers of Combining Systems (FROCOS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, 1996.