Vers une automatisation de la certification des propriétés de clôture pour Prolog

Thierry Marianne, Fred Mesnard et Etienne Payet

Université de La Réunion

L'interprétation abstraite et la preuve interactive de programmes se situent aux deux extrêmes du spectre des possibilités d'automatisation appliquée à la vérification de programmes. Notre objectif est de préserver certains de leurs avantages respectifs en combinant ces deux techniques.

Pour concrétiser cet objectif, nous avons choisi l'analyse de clôture de la programmation logique (*groundness analysis* en anglais). Dans un premier temps, nous calculons les propriétés de clôture des arguments d'un programme logique sous forme de formules booléennes obtenues par interprétation abstraite. Remarquons que ces invariants sont calculés par des algorithmes et des librairies non certifiés. En implantant un algorithme prouvant la complétude de la déduction naturelle du calcul des propositions, nous générons une dérivation logique pour chaque invariant. Nous construisons ces dérivations dans la syntaxe de LPTP (*Logic Program Theorem Prover*), un assistant de preuve en déduction naturelle pour les programmes Prolog conçu par Robert F. Stärk au milieu des années 90. Finalement, nous validons *a fortiori* chaque dérivation par le vérificateur de preuve de LPTP. Nous automatisons ainsi la certification formelle d'invariants de clôture générés par interprétation abstraite.

Nous présentons une expérimentation visant à évaluer cette méthodologie en l'appliquant à un jeu de programmes logiques.

1 Introduction

Les performances de l'exécution des programmes logiques peuvent être optimisées à partir des résultats de l'analyse statique et de l'interprétation abstraite [CC77][Bru01]. Lorsque des atomes ne partagent aucune variable, il devient possible d'introduire du parallélisme dans la résolution de sous-buts indépendants [MH89]. Cependant la vérification de l'indépendance des variables à l'exécution est coûteuse, ce qui diminue l'intérêt du parallélisme. L'élimination de ces vérifications est possible en collectant des informations portant sur la clôture des variables à la compilation.

L'analyse du comportement d'un programme logique par interprétation abstraite [CC92a] peut s'appuyer sur la clôture des variables comme abstraction possible des substitutions appliquées à l'exécution du programme. Nous supposons qu'une stratégie d'évaluation de gauche à droite des littéraux ait été sélectionnée pour chaque clause d'un programme logique donné analysé. À la sélection d'une clause pour résolution d'un but donné, ses variables sont renommées par application d'une substitution de renommage des variables.

Définition 1 (Terme)**.** Les termes sont définis de manière inductive tel que suit :
— une variable est un terme
— si f est un symbole de fonction n-aire et $t_1, \ldots, t_n$ sont des termes alors $f(t_1, \ldots, t_n)$ est un terme. En particulier, toute constante est un terme pour un symbole de fonction d'arité 0.

Définition 2 (Littéral)**.** Un littéral est un terme ou la négation d'un terme.

Définition 3 (Substitution)**.** Une substitution est une application finie qui affecte à chaque variable x de son domaine un terme t différent de x. On écrit $\{x_1/t_1, \ldots, x_n/t_n\}$ où
— $x_1, \ldots, x_n$ sont des variables distinctes,
— $t_1, \ldots, t_n$ sont des termes,
— Pour tout $i \in [1, n], x_i \neq t_i$

Définition 4 (Terme clos)**.** Un terme t est clos si pour toute substitution σ, $t\sigma$ ne contient aucune variable. Par exemple, le terme constant de la liste vide [] est clos.

Dans les années 90, par définition de sa théorie de vérification de programmes Prolog, Robert F. Stärk démontre que la clôture d'un terme t dans son framework IND(P) implique que ce terme t est clos [Sta98]. IND(P) consiste en un ensemble d'axiomes du premier ordre associé au programme logique P, défini comme son extension inductive. Cette théorie inclut la complétion de Clark [Cla78] et l'induction d'après la définition de ses prédicats. Cette garantie nous permet de formuler des propriétés de clôture de P dans le langage de spécification de LPTP (*Logic Program Theorem Prover*), dont la sémantique est celle du calcul des prédicats de la logique classique [Sta98]. Nous obtenons ces propriétés à partir de conditions suffisantes de terminaison inférées par interprétation abstraite. En pratique, nous calculons ces inférences avec cTI (*constraint-based Termination Inference*) [MN01a], avant de prouver la correction des propriétés qui en découlent en les certifiant avec LPTP.

Cet article s'organise comme suit. La section 2 fait état de techniques d'analyse par interprétation abstraite appliquées aux programmes logiques en vue d'obtenir des relations de clôture inter-arguments sous forme booléenne. Nous illustrons cette application à partir de l'analyse d'un programme d'inversion d'une liste faisant intervenir deux prédicats *append/3* et *reverse/2*. La section 3 décrit comment nous formulons des propriétés de clôture découlant directement d'invariants obtenus par interprétation abstraite. Nous y présentons un algorithme de construction de la dérivation de ces propriétés en déduction naturelle dans le cadre théorique de LPTP. Pour ce faire, nous explicitons des éléments de la preuve par induction d'une formule propositionnelle proposée par Huth et Ryan [HR00]. En dernier lieu, nous présentons un tableau comparatif pour différents programmes logiques des résultats recueillis par application de cette méthodologie.

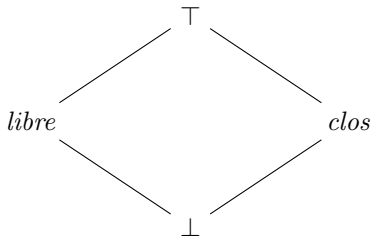
2 Analyse de clôture par interprétation abstraite

L'interprétation abstraite est une méthode formelle de conception d'approximations de la sémantique d'un programme servant à collecter des informations pour l'analyse du flot des données pour des domaines concrets qui peuvent être non bornés à l'exécution du programme. Le résultat de cette analyse statique est une abstraction du comportement du programme durant son exécution [CC92b]. L'interprétation abstraite a été étendue à l'analyse de flot des programmes logiques et formalisée dans le cadre de l'inférence de modes en particulier [Bru01].

Définition 5 (Mode d'un prédicat)**.** Un mode d'un prédicat dans un programme logique indique comment ses arguments seront instanciés à l'appel de ce prédicat. Les arguments sont ainsi classés par leur appartenance aux classes d'équivalence suivantes après application des substitutions : l'ensemble vide (*vide*), l'ensemble des termes clos (*clos*), l'ensemble des variables libres (*libre*) et l'ensemble de tous les termes (*inconnu*).

En analyse de clôture, les modes des prédicats appelés sont des approximations du domaine concret de l'exécution d'une requête [Mel86]. L'appartenance aux ensembles *vide*, *clos*, *libre* et *inconnu* est l'abstraction du nombre des termes liés aux arguments des prédicats, à partir de laquelle une exécution approchée du programme peut être calculée [DW88].

Un interprète abstrait met en correspondance l'ensemble des substitutions appliquées à l'exécution pour tous les appels possibles des prédicats du programme aux substitutions abstraites du treillis fini suivant où $\top$ et $\bot$ sont respectivement l'ensemble de tous les termes (*inconnu*) et l'ensemble vide (*vide*).



Les dépendances entre la clôture des variables peuvent être représentées par des approximations sous forme de fonctions booléennes tels que $Bfun_n$ [AMSS98] [MS93].

Définition 6 ($Bfun_n$, Fonctions booléennes)**.** Pour tout entier naturel n, une fonction booléenne n-aire est une fonction $Bool^n \to Bool$, où $Bool$ est un domaine booléen. L'ensemble de toutes les fonctions booléennes n-aire est dénoté $Bfun_n$. Il est ordonné par implication sémantique ($\models$).

Une relation de dépendance de clôture telle que « lorsqu'une variable Y d'un programme logique est instanciée à un terme clos, alors une variable X devient un terme clos à son tour » peut être dénotée par la fonction booléenne $y \to x$ où y et x sont des variables propositionnelles correspondant à des approximations des états d'instanciations respectifs de Y et X.

Dans le contexte de l'implantation de cTI (*constraint-based Termination Inference*) [MN01b], une forme d'interprétation abstraite nommée *compilation abstraite* aboutit sur des approximations du programme analysé afin d'obtenir des conditions suffisantes de terminaison universelle. Ces approximations pour le programme P comme relations de clôture inter-arguments exprimées sous forme booléenne serviront à formuler des hypothèses de propriétés de clôture du programme à prouver et dont nous certifierons les dérivations.

Prenons comme exemple de programme logique P le programme *reverse* ci-dessous. Ce programme est constitué de deux prédicats *reverse/2* et *append/3*. L'appel à *append/3* réussit dans le cas de la satisfaction de sa première clause lorsque son premier argument est une liste vide et que ses deuxième et troisième arguments sont égaux. Lorsqu'une liste non vide `[X|Xs]` est passée en premier argument d'*append/3*, l'appel réussit si la relation `append(Xs, Ys, Zs)` est vraie par appel récursif à *append/3* c.a.d que `Zs` résulte de la concaténation de `Xs` et `Ys`. La récursion s'arrête lorsque la première clause est satisfaite. L'appel à *reverse/2* réussit quand ses deux arguments sont des listes vides et que sa première clause est satisfaite. Dans le cas de listes non vides passées en argument, la résolution de sa seconde clause réussit quand son second argument `Ys` est le résultat de la concaténation d'une liste intermédiaire `Zs` et de la liste `[X]` comprenant la tête du premier argument passée à la seconde clause de *reverse/2*. `Zs` doit alors être le résultat de l'inversion de la queue du premier argument `Xs` par appel récursif à *reverse/2*. Cette récursion est interrompue lorsque la première clause est satisfaite.

```
append([], Xs, Xs).
append([X|Xs], Ys, [X|Zs]) :-
  append(Xs, Ys, Zs).
```

```
reverse([], []).
reverse([X|Xs], Ys) :-
    reverse(Xs, Zs),
    append(Zs, [X], Ys).
```

L'analyse de P avec cTI donne les relations de dépendance de clôture des arguments sous les formes booléennes suivantes :

$$(x_{append} \wedge y_{append}) \leftrightarrow z_{append} \quad (1)$$
$$x_{reverse} \leftrightarrow y_{reverse} \quad (2)$$

Ces formules indiquent respectivement que le dernier argument z_{append} du prédicat *append/3* est clos si et seulement son premier argument x_{append} et son deuxième argument y_{append} sont clos et que la clôture du premier argument $x_{reverse}$ de *reverse/2* est équivalente à la clôture de son second argument $y_{reverse}$.

Les modes des prédicats de notre programme peuvent être listés de manière informelle en substituant les classes d'équivalence de la **Définition 5** aux arguments des prédicats à partir des relations (*1*) et (*2*) :

```
append(clos, clos, clos).          reverse(clos, clos).
append(libre, clos, libre).        reverse(libre, libre).
append(clos, libre, libre).
append(libre, libre, libre).
```

3 Certification de propriétés de clôture avec LPTP

Nous considérons une formule de propriété de clôture fournie par cTI pour un programme logique pur contenant des procédures définies sous forme de prédicats. Nous avons montré comment le schéma d'*axiome VIII* du cadre théorique IND(P) de LPTP [Sta98] nous autorise à générer un axiome d'induction pour un prédicat directement récursif défini par l'utilisateur et une formule à prouver [MMP24].

Par exemple, démontrons le lemme **append3_gr** obtenu par interprétation abstraite en fin de section précédente où **S append/3** signifie qu'un appel au prédicat *append/3* réussit pour toutes variables x_1, x_2 et x_3 déclarées de manière universelle et qui lui sont passées en argument. Les extraits de preuve de notre exemple de référence sont exportés directement à partir de LPTP par l'assistant au format TEX.

Lemma [append3_gr] $\forall x_1, x_2, x_3\, (\mathbf{S}\,\mathbf{append}(x_1, x_2, x_3) \rightarrow (gr(x_3) \wedge gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge gr(x_2) \wedge \neg gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge \neg gr(x_1)))$.

Dans le format de LPTP, les propriétés de clôture sont exprimées en appelant le prédicat unaire *gr* exprimant qu'un terme est clos avec la définition suivante.

Définition 7 (Prédicat unaire *gr*)**.** Soient s et t des termes,
R un prédicat non-prédéfini n-aire où n est un entier naturel,
$t_1, \ldots, t_n$ des termes passés en argument de R, et G et H des buts
Les constantes *true* et *fail* sont closes :
$gr(true) := \top \qquad gr(fail) := \top$
La clôture d'une unification équivaut à la conjonction de celle de ses membres :
$gr(s = t) := gr(s) \wedge gr(t)$
La clôture d'un appel à R équivaut à la conjonction de celle de ses arguments :
$gr(R(t_1, \ldots, t_n)) := gr(t_1) \wedge \cdots \wedge gr(t_n)$,
La clôture d'une conjonction de buts équivaut à la clôture simultanée de ces buts :
$gr((G, H)) := gr(G) \wedge gr(H)$
La clôture d'une disjonction de buts équivaut à la clôture simultanée de ces buts :
$gr((G; H)) := gr(G) \wedge gr(H)$
L'existence d'une variable x pour laquelle un but G est clos équivaut à la clôture d'une restriction de ce but après substitution par x d'une variable fraîche y, n'apparaissant pas dans *some x G*

$gr(some\ x\ G) := \exists x\ gr(G)$
La clôture de la négation d'un but équivaut à la clôture de ce but :
$gr(not\ G) := gr(G)$

L'application de l'*axiome II* de IND(P) [Sta98] (faisant partie intégrante de LPTP) permet d'introduire des équivalences portant sur le prédicat gr appliqué aux termes composés.

Axiome II de IND(P) pour *gr*
4. $gr(c)$ [si c est une constante]
5. $gr(x_1) \wedge \ldots gr(x_m) \Leftrightarrow gr(f(x_1, \ldots, x_m))$ [si f est m-aire]

LPTP permet de prouver des propriétés opérationnelles d'un programme logique. En particulier, l'opérateur déclaratif **S**ucceeds exprime le succès de l'appel du prédicat auquel il est appliqué [Sta98]. La traduction automatique à partir du code Prolog des prédicats *append/3* et *reverse/2* en des définitions inductives positives par LPTP donne les équivalences :

$\forall x, y, z\,(\mathbf{S}\,\mathbf{append}(x, y, z) \leftrightarrow (x = [\,] \wedge z = y) \vee$
$(\exists v_0, xs, zs\,(x = [v_0|xs] \wedge z = [v_0|zs] \wedge \mathbf{S}\,\mathbf{append}(xs, y, zs))))$

$\forall x, y\,(\mathbf{S}\,\mathbf{reverse}(x, y) \leftrightarrow (x = [\,] \wedge y = [\,]) \vee$
$(\exists v_0, xs, zs\,(x = [v_0|xs] \wedge \mathbf{S}\,\mathbf{reverse}(xs, zs) \wedge \mathbf{S}\,\mathbf{append}(zs, [v_0], y))))$

La forme générale de la formule de clôture obtenue par interprétation abstraite à certifier est une implication dont la conclusion est en forme normale disjonctive :

$$\forall x_1, \ldots, x_i\ \mathbf{S}R(t_1, \ldots, t_j) \implies \bigvee_{k=1}^{Dis} \bigwedge_{m \in Con_k} gr(x_m)$$

où R est un prédicat non-prédéfini n-aire, $t_1, \ldots, t_j$ les arguments passés à R lorsqu'il réussit (termes composés des variables $x_1, \ldots, x_i$), i, j, k, m des entiers naturels, Dis le nombre de disjonctions et Con_k l'ensemble des variables dont les propriétés de clôture sont exprimées par le k-ème disjoint. Afin de démontrer chaque étape du schéma d'induction généré à partir de cette formule, nous implémentons un algorithme inspiré de la preuve de la proposition suivante (qui intervient dans la preuve de complétude de la logique propositionnelle donnée dans [HR00]).

Proposition 1 (Proposition 1.38 de [HR00])**.** *Soit une formule ϕ avec $p_1, p_2, \ldots, p_n$ ses atomes propositionnels. Soit l toute ligne du tableau de vérité de ϕ. Pour tout $1 \leq i \leq n$, soit $\hat{p}_i$ égal à p_i si l'entrée de p_i à la ligne l est T, autrement $\hat{p}_i$ est $\neg p_i$. Nous avons alors*

1. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi$ peut être démontrée si l'entrée pour ϕ à la ligne l est T.*
2. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi$ peut être démontrée si l'entrée pour ϕ à la ligne l est F.*

La preuve de cette proposition est effectuée par induction structurelle de la formule ϕ.

1. Si ϕ est un atome propositionnel p, alors on doit montrer que $p \vdash p$ et $\neg p \vdash \neg p$. Ces preuves tiennent en une ligne.
2. Si ϕ est de la forme $\neg\phi_1$, nous avons encore deux cas à considérer. Dans un premier temps, nous supposons que ϕ s'évalue à T, alors ϕ_1 s'évalue à F. En effet, ϕ_1 possède les mêmes atomes propositionnels que ϕ. On peut donc appliquer l'hypothèse d'induction afin de conclure que $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi_1$, mais $\neg\phi_1$ est juste ϕ et la preuve est donc terminée. Dans un second temps, si ϕ s'évalue à F, alors ϕ_1 s'évalue à T et nous obtenons $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ par induction. Par application de la règle $\neg\neg i$, on pourrait étendre la preuve de $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ d'une ligne à $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\neg\phi_1$ mais $\neg\neg\phi_1$ est juste $\neg\phi$.

Tous les cas restants présentent deux sous-formules où ϕ est de la forme $\phi_1 \circ \phi_2$ avec $\circ$ un connecteur logique parmi $\{\implies, \vee, \wedge\}$. Soient $q_1, \ldots, q_l$ les atomes propositionnels de

ϕ_1 et $r_1, \ldots, r_k$ les atomes propositionnelles de ϕ_2. Nous avons $\{q_1, \ldots, q_l\} \cup \{r_1, \ldots, r_k\} = \{p_1, \ldots, p_n\}$. Par conséquent, lorsque nous avons prouvé que $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ et $\hat{r}_1, \ldots, \hat{r}_k \vdash \phi_2$ alors $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ par la règle $\wedge i$. Afin de prouver la conclusion désirée pour ϕ (ou $\neg\phi$), nous appliquons notre hypothèse d'induction à partir de conjonctions à démontrer pour chacune des valeurs de vérité possiblement prises par les sous-formules.

Pour le cas où ϕ est de la forme $\phi_1 \wedge \phi_2$, nous avons quatre sous-cas à traiter.

1. Quand ϕ_1 et ϕ_2 s'évaluent à T, alors l'hypothèse d'induction donne $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ et $\hat{r}_1, \ldots, \hat{r}_k \vdash \phi_2$, et par conséquent $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$.
2. Quand ϕ_1 s'évalue à F et ϕ_2 s'évalue à T (resp. ϕ_1 s'évalue T et ϕ_2 s'évalue à F), alors l'hypothèse d'induction et la règle $\wedge i$ donnent $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ (resp. $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$) et nous devons prouver $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$ (resp. $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$). Supposons $\neg(\neg\phi_1 \wedge \phi_2)$, nous obtenons $\phi_1 \vee \neg\phi_2$ par application de la loi de De Morgan. Deux cas se présentent par application de la règle $\vee e$. Dans un cas, nous obtenons ϕ_1, ce qui contredit que ϕ_1 s'évalue à F par hypothèse et dans l'autre cas, nous obtenons $\neg\phi_2$, ce qui contredit que ϕ_2 s'évalue à T par hypothèse. Nous avons montré $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$ par contradiction. On peut montrer $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$ en appliquant un raisonnement similaire.
3. Et quand ϕ_1 et ϕ_2 s'évaluent à F, notre hypothèse d'induction et la règle $\wedge i$ nous permettent de conclure $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$, et nous devons prouver $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$. Supposons $\neg\neg(\phi_1 \wedge \phi_2)$. Par application de la règle $\neg\neg e$, nous obtenons $\phi_1 \wedge \phi_2$. Par application de la règle $\wedge e$, nous obtenons que ϕ_1 s'évalue à T, ce qui contredit l'hypothèse selon laquelle ϕ_1 s'évalue à F et dans l'autre cas, nous obtenons que ϕ_2 s'évalue à T, ce qui contredit l'hypothèse selon laquelle ϕ_2 s'évalue à F. Nous avons démontré $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$ par contradiction.

Dans chacun des cas où ϕ est de la forme $\phi_1 \vee \phi_2$ ou ϕ est de la forme $\phi_1 \implies \phi_2$, quatre autres sous-cas peuvent être traités de manière similaire.

En nous inspirant des étapes de la preuve par induction de la Proposition 1.38, nous substituons des appels au prédicat unaire *gr* aux variables propositionnelles de la formule de clôture. Comme LPTP est implémentée en ISO-Prolog, nous réutilisons sa propre syntaxe, ses opérateurs déclaratifs de succès de prédicats et ses tactiques afin de mécaniser le processus de génération d'un terme de preuve complet.

Tout d'abord, nous énumérons les n variables distinctes de la propriété de clôture cible afin d'assembler une preuve correspondant aux 2^n lignes d'un tableau de vérité pour toutes les variables à prendre en compte. Nous supposons que la propriété de clôture cible est toujours vraie car elle a été inférée avec cTI comme invariant de notre programme sous forme booléenne. Cette preuve existe par complétude du calcul des propositions (Théorème 1.37 proposé par Huth et Ryan [HR00]).

Par conséquent, il est possible de découvrir des dérivations à partir d'invariants produits par cTI. En effet, pour nos besoins quant aux propriétés de clôture des variables de notre programme, les relations de clôture inférées pour chacun des prédicats suffisent.

Nous avons implémenté la procédure `prove_with_premises` en pseudo-code ci-dessous. Elle retourne une dérivation (`Deriv`) à partir d'une formule de clôture (`Phi`), de prémisses (`Premises`) et de la valeur de vérité (`TruthValue`) des différentes relations (clôture `gr`, négation `~`, conjonction `&`, disjonction `\/`, et implication `=>`) à démontrer par appel récursif de cette procédure pour des sous-formules de `Phi` (`Form`, `Phi1` et `Phi2`).

```
prove_with_premises(Phi, TruthValue, Premises)
    switch Phi
        case gr(Form) : ... return Deriv
        case ~ Form  : ... return Deriv
        case Phi1 & Phi2 : ... return Deriv
        case Phi1 \/ Phi2 : ... return Deriv
        case Phi1 => Phi2 : ... return Deriv
```

L'application d'éliminations de disjonctions systématique à partir des 2^n cas obtenus se traduit directement en LPTP par l'application récursive de la tactique *cases* pour les disjonctions exprimant qu'une variable est close ou libre. La formule de l'invariant est réduite par décomposition syntaxique. Nous prouvons toutes les sous-formules et leurs combinaisons pré-requises une à une afin de pouvoir reconstruire l'invariant. Lorsqu'on réduit l'invariant par lequel nous concluons nos dérivations, les séquents qui dépendent uniquement des prémisses en provenance des applications successives du principe du tiers exclus sont progressivement mis bout à bout afin de constituer une preuve complète qui couvre les 2^n cas de clôture des variables en jeu.

L'axiome du schéma d'induction pour le prédicat non-prédéfini *append/3* et la formule de clôture fournie par cTI permettent de générer le lemme **append3_gr** avec deux trous (**GAP**) correspondant au cas de base et à l'étape d'induction à prouver. Afin de combler ces trous, l'algorithme cité précédemment est mis en œuvre pour chacune des deux étapes du schéma d'induction. Dans les deux cas, nous cherchons à prouver une tautologie (une proposition vraie quelque soit la clôture des variables) sous forme normale disjonctive comme relation de clôture d'arguments pour le prédicat *append/3*.

En considérant le cas de base en premier lieu (entre les deux lignes horizontales), nous observons que $gr([])$ est vrai du fait de l'*axiome II.4* de IND(P). En énumérant les variables du cas de base, nous découvrons une variable unique. Seuls deux cas sont à considérer selon la clôture de x_4. L'algorithme décrit auparavant nous donne l'extrait ci-dessous après avoir rempli le trou correspondant au cas de base entre les lignes horizontales.

Lemma [append3_gr] $\forall x_1, x_2, x_3\,(\mathbf{S}\,\mathbf{append}(x_1, x_2, x_3) \rightarrow (gr(x_3) \wedge gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge gr(x_2) \wedge \neg gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge \neg gr(x_1)))$.
Proof.

Induction$_0$: $\forall x_1, x_2, x_3\,(\mathbf{S}\,\mathbf{append}(x_1, x_2, x_3) \rightarrow (gr(x_3) \wedge gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge gr(x_2) \wedge \neg gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge gr(x_1)) \vee (\neg gr(x_3) \wedge \neg gr(x_2) \wedge \neg gr(x_1)))$.

Hypothesis$_1$: none.
Case$_2$: $gr(x_4)$. $gr(x_4) \wedge gr(x_4)$. $gr(x_4) \wedge gr(x_4) \wedge gr([])$.
$gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([])$.
$gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([])$.
$gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge \neg gr([])$.
Case$_2$: $\neg gr(x_4)$. $\neg gr(x_4) \wedge \neg gr(x_4)$. $\neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([])$.
$gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([])$.
$gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge \neg gr([])$.
Hence$_2$, in all cases : $gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge \neg gr([])$.
Conclusion$_1$: $gr(x_4) \wedge gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge gr(x_4) \wedge \neg gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge gr([]) \vee \neg gr(x_4) \wedge \neg gr(x_4) \wedge \neg gr([])$.

Hypothesis$_1$: $gr(x_8) \wedge gr(x_7) \wedge gr(x_6) \vee \neg gr(x_8) \wedge gr(x_7) \wedge \neg gr(x_6) \vee \neg gr(x_8) \wedge \neg gr(x_7) \wedge gr(x_6) \vee \neg gr(x_8) \wedge \neg gr(x_7) \wedge \neg gr(x_6)$ and $\mathbf{S}\,\mathbf{append}(x_6, x_7, x_8)$.
Case$_2$: $gr(x_5)$.
Case$_3$: $gr(x_6)$.
Case$_4$: $gr(x_7)$.
Case$_5$: $gr(x_8)$. $\bot$ by **GAP**.
Hence$_5$, in all cases : $(gr(x_8) \wedge gr(x_7) \wedge gr(x_6)) \vee (\neg gr(x_8) \wedge gr(x_7) \wedge \neg gr(x_6)) \vee (\neg gr(x_8) \wedge \neg gr(x_7) \wedge gr(x_6)) \vee (\neg gr(x_8) \wedge \neg gr(x_7) \wedge \neg gr(x_6)) \rightarrow (gr([x_5|x_8]) \wedge gr(x_7) \wedge gr([x_5|x_6])) \vee (\neg gr([x_5|x_8]) \wedge gr(x_7) \wedge \neg gr([x_5|x_6])) \vee (\neg gr([x_5|x_8]) \wedge \neg gr(x_7) \wedge gr([x_5|x_6])) \vee (\neg gr([x_5|x_8]) \wedge \neg gr(x_7) \wedge \neg gr([x_5|x_6]))$.

$\text{Conclusion}_4 : (gr([x_5|x_8]) \wedge gr(x_7) \wedge gr([x_5|x_6])) \vee$
$(\neg gr([x_5|x_8]) \wedge gr(x_7) \wedge \neg gr([x_5|x_6])) \vee (\neg gr([x_5|x_8]) \wedge \neg gr(x_7) \wedge gr([x_5|x_6])) \vee$
$(\neg gr([x_5|x_8]) \wedge \neg gr(x_7) \wedge \neg gr([x_5|x_6]))$. □

Pour l'étape d'induction, nous appliquons le même algorithme afin de dériver la formule à prouver après en avoir énuméré les variables (x_5, x_6, x_7 et x_8). Par souci de concision, seul l'un des trous sous-jacent à l'étape d'induction à remplir a été préservé après avoir fixé des cas de clôture pour chacune des quatre variables à considérer. Les dérivations complètes sont disponibles de manière publique en ligne [1]. En vérifiant le terme de preuve généré avec LPTP, nous certifions la validité des dérivations ainsi obtenues de manière automatique.

En considérant enfin *reverse/3* comme prédicat cible pour dériver la seconde formule de clôture inférée par cTI, nous réutilisons le lemme associé aux prédicats dont reverse/3 dépend, c'est-à-dire que la propriété de clôture d'*append/3* est utilisé comme prémisse supplémentaire avant d'appliquer l'algorithme de construction de preuve.

4 Évaluation expérimentale

4.1 Cadre expérimental

Prog.	Prop.	Vars	Inf. (ms)	Deriv. (ms)	Cert. (ms)	LOC
member.pl	1	3	2.5	5.2	6.9	269
ackermann.pl	1	3	3.1	7.9	15.2	949
for.pl	2	4	2.7	7.9	6.1	273
addmul.pl	2	6	3.0	8.5	16.2	911
fib.pl	2	5	3.2	9.1	14.2	970
nat.pl	4	8	3.7	10.3	16.8	975
micgram.pl	3	9	3.9	12.9	36.8	680
split.pl	1	3	6.9	14.0	28.9	1123
int.pl	6	11	5.6	17.0	19.3	1049
average1.pl	3	7	4.6	17.9	119.5	8846
suffix.pl	4	10	7.9	18.9	23.2	1203
list.pl	5	12	12.4	28.5	66.6	2873
apprev.pl	2	5	7.7	33.4	57.3	2422
permutation.pl	7	19	13.0	40.7	241.6	9335
mc_pera.pl	4	9	9.9	40.8	83.9	3916
testapp3.pl	4	15	9.4	45.1	253.4	10019
reverse.pl	4	10	11.8	48.4	95.3	3958
derivDLS.pl	1	3	11.9	304.9	152.5	3931
trans.pl	2	5	27.3	306.6	235.6	9456
sort.pl	9	22	41.2	351.7	12413.5	71067
transitiveclosure.pl	6	18	24.1	648.2	676.8	14213
mergesort.pl	4	10	31.1	41366.8	1463.2	48400

Nous avons expérimenté la démarche présentée à partir de la bibliothèque de LPTP, à laquelle d'autres programmes ont été ajoutés jusqu'à atteindre un total de trente cinq programmes. Nous avons collecté les résultats ci-avant triés par temps de construction de dérivation croissant (colonne Deriv.) avec la version 9.2.2 de SWI-Prolog et un MacBook Pro (Apple M2, MacOS Sonoma 14.6.1).

Nous avons mesuré pour chaque programme (colonne Prog.) le nombre de propriétés de clôture (colonne Prop.), le temps nécessaire afin d'obtenir les invariants portant sur les relations de clôture inférées par cTI (colonne Inf.). Nous avons ensuite mesuré pour chaque programme le temps nécessaire pour construire la preuve des propriétés de clôture associées (colonne Deriv.) et le temps pour certifier les dérivations calculées avec LPTP

1. https ://github.com/atp-lptp/lptp

(colonne Cert.). Le nombre de lignes des preuves générées est donné à titre indicatif (colonne LOC). Nous nous appuyons sur un prédicat appartenant à LPTP pour formater les preuves (*prt__write/1*). Le surcoût associé à ce formatage a été exclu du temps de construction de dérivation. L'implantation complète permettant de générer le tableau ci-dessus est disponible en ligne [2].

4.2 Limites

Nous observons une complexité de génération des cas de dérivation à traiter d'ordre exponentiel en le nombre de variables à prendre en considération dans chacune des formules de clôture inférées par interprétation abstraite. Cette complexité est l'une des principales limites de notre implantation, due au choix de l'algorithme de Huth et Ryan construisant une déduction naturelle pour toute tautologie. Nous considérons qu'un démonstrateur automatique tel que *Vampire* pourrait nous permettre de générer une preuve plus courte avant de la traduire au format LPTP [MMP24].

Par ailleurs, nous prévoyons d'étendre la méthodologie appliquée en prenant en compte les clauses mutuellement récursives. Par exemple, dans le cadre de l'analyse du programme *even_odd* ci-après, la clôture de l'argument passé à l'appel de la seconde clause du prédicat *even/1* est équivalente à celle de l'argument passé au prédicat *odd/1* dans le corps de cette clause. De plus, la clôture de l'argument passé à l'appel du prédicat *odd/1* est équivalente à celle de l'argument passé à *even/1*. Une étape supplémentaire de combinaison des formules de clôture de ces prédicats est donc nécessaire afin de les dériver en même temps.

```
even(0).                        odd(s(X)) :- even(X)
even(s(X)) :- odd(X).
```

Une autre source d'échec provient de l'absence de gestion des regroupements de prédicats en modules. Nous pourrons importer les clauses absentes des programmes analysés avant construction des dérivations par analyse du graphe d'appel.

5 Conclusion

Indépendamment de l'interprétation abstraite, les assistants de preuve tels qu'Isabelle avec *Sledgehammer* s'appuient sur des outils comme *cvc5* [3] qui permettent d'aller plus vite dans ses démonstrations en recevant des suggestions de lemmes ou de tactiques. Afin de préserver la possibilité de vérifier les preuves entièrement, un mécanisme de reconstruction de preuve a été introduit à partir de certificats [FS19].

Dans cet article, nous avons illustré cette idée dans le cadre de la programmation logique. Nous certifions via le vérificateur intégré à LPTP des propriétés de clôture en construisant automatiquement les preuves correspondant à ces propriétés de clôture. Nous combinons ainsi l'avantage de la génération algorithmique d'invariants par interprétation abstraite à celui de la vérification formelle offert par un outil interactif de preuve.

Nous espérons réduire prochainement les temps de calcul de l'approche exposés à la section précédente. D'autres types de relations inter-arguments, correspondant à d'autres domaines abstraits et d'autres procédures de preuve, font partie des voies à explorer pour montrer la généricité de notre approche.

Remerciements

Nous remercions Manuel Hermenegildo, Pedro Lopez et José Morales pour les nombreuses discussions autour ce thème.

2. https ://github.com/atp-lptp/lptp
3. https ://cvc5.github.io/blog/2024/03/15/isabelle-reconstruction.html

Références

[AMSS98] Tania Armstrong, Kim Marriott, Peter Schachte et Harald Søndergaard : Two classes of boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998. Selected Papers of the First International Static Analysis Symposium.

[Bru01] Maurice Bruynooghe : *A framework for the abstract interpretation of logic programs*. Department of Computer Science, K.U.Leuven, Leuven, Belgium, 1987-10-01.

[CC77] Patrick Cousot et Radhia Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CC92a] P. Cousot et R. Cousot : Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. see `http://www.di.ens.fr/~cousot`.).

[CC92b] Patrick Cousot et Radhia Cousot : Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, août 1992. Funding Information : We would like to thank Alan Mycroft for numerous judicial and useful comments on a first draft of this paper as well as the anonymous referees for their constructive suggestions for improvement. This work was supported in part by Esprit BRA 3124 Semantique and CNRS PRC C3.

[Cla78] Keith L. Clark : *Negation as Failure*, pages 293–322. Springer US, Boston, MA, 1978.

[DW88] Saumya K. Debray et David S. Warren : Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207–229, 1988.

[FS19] Mathias Fleury et Hans-Jörg Schurr : Reconstructing verit proofs in isabelle/hol. *Electronic Proceedings in Theoretical Computer Science*, 301:36–50, août 2019.

[HR00] M. Huth et M. Ryan : *Logic in Computer Science : Modelling and Reasoning about Systems*. Logic in Computer Science : Modelling and Reasoning about Systems. Cambridge University Press, 2000.

[Mel86] C. S. Mellish : Abstract interpretation of prolog programs. *In* Ehud Shapiro, éditeur : *Third International Conference on Logic Programming*, pages 463–474, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[MH89] Kalyan Muthukumar et Manuel V. Hermenegildo : Determination of variable dependence information through abstract interpretation. *In NACLP*, 1989.

[MMP24] Fred Mesnard, Thierry Marianne et Etienne Payet : Automated theorem proving for prolog verification. *In* Nikolaj Bjørner, Marijn Heule et Andrei Voronkov, éditeurs : *LPAR 2024 Complementary Volume*, volume 18 de *Kalpa Publications in Computing*, pages 137–151. EasyChair, 2024.

[MN01a] Fred Mesnard et Ulrich Neumerkel : Applying static analysis techniques for inferring termination conditions of logic programs. *In* Patrick Cousot, éditeur : *Static Analysis*, pages 93–110, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[MN01b] Frédéric Mesnard et Etienne Neumerkel, Ulrich et Payet : cTI : un outil pour l'inférence de conditions optimales de terminaison pour Prolog. *In 10eme Journées francophones de programmation logique et programmation par contraintes (JFPLC'2001)*, pages 271–286, Paris, France, avril 2001. Association Française pour

la Programmation en Logique et la programmation par Contraintes (AFPLC), Hermes Science Publications.

[MS93] Kim Marriott et Harald Søndergaard : Precise and efficient groundness analysis for logic programs. *ACM Lett. Program. Lang. Syst.*, 2(1–4):181–196, mars 1993.

[Sta98] Robert F. Staerk : The theoretical foundations of lptp (a logic program theorem prover). *The Journal of Logic Programming*, 36(3):241–269, 1998.