Dessine moi un domaine abstrait fini

une recette à base de Camlp4 et de solveurs SMT

Pierre Roux [*, **] **, Pierre-Loïc Garoche** [**]

[*] *ENS Lyon,*
pierre.roux@ens-lyon.org
[**] *Onera/DTIM/ISC,*
pierre-loic.garoche@onera.fr

Résumé. *Parmi les outils de vérification utilisés aujourd'hui, les interprètes abstraits jouent un rôle de plus en plus important. Ils reposent sur un représentation abstraite des propriétés des logiciels, les domaines abstraits. Dans l'enseignement de ces techniques, on veut pouvoir synthétiser totalement ou partiellement des analyseurs pour faire intervenir les élèves et la génération automatique de domaines abstraits finis prend alors tout son sens. Nous présentons ici notre approche qui génère un domaine abstrait et ses fonctions de transfert sémantique à partir d'une description de la structure de son treillis et d'une fonction de concrétisation. Cette méthode repose sur Camlp4 pour construire un module Caml de type « domaine abstrait » et sur des solveurs SMT pour calculer au mieux les opérations de transfert du domaine.*

Mots-Clés : *interprétation abstraite, génération automatique, camlp4, solveurs SMT*

Studia Informatica Universalis.

1. Contexte

Les programmes informatiques sont aujourd'hui omniprésent au sein de nombreux objets de notre vie quotidienne. Du bon fonctionnement de certains de ces logiciels, dits critiques, peut dépendre la vie de leur utilisateur. C'est le cas par exemple des programmes de contrôle de certains équipements médicaux ou des commandes de vol d'un avion de ligne. D'où l'intérêt porté aux outils d'analyse statique permettant de prouver la conformité de tels programmes à certaines spécifications et donc à l'enseignement de ces techniques.

D'autre part, faire écrire un analyseur « jouet » est certainement un excellent moyen d'enseigner ces méthodes. Il est alors intéressant d'essayer de simplifier la réalisation des parties les plus fastidieuses pour les élèves, leur permettant ainsi de se concentrer sur les aspects les plus importants.

Nous présentons donc un générateur de code permettant de réaliser des parties de tels analyseurs constituées de code particulièrement répétitif, parfois volumineux et ainsi enclines à de fréquentes erreurs lors de leur programmation. On verra comment l'on peut ainsi remplacer un code OCaml [OC] comprenant plusieurs milliers de motifs de filtrage par une description graphique de quelques dizaines de lignes. Ceci à l'aide de Camlp4 [Cp4] pour la génération de code OCaml et de solveurs SMT [NOT06, dMB08, DdM06, CCKL08] pour le calcul de sémantiques abstraites.

Après une rapide présentation de quelques éléments de théorie utilisés par la suite (section 2), les sections 3 et 4 présentent le langage d'entrée de l'outil et l'utilisation faite de Camlp4. Le calcul de sémantiques abstraites effectués à l'aide de solveurs SMT occupe les trois suivantes, la section 8 présentant enfin un exemple – presque – complet tandis que la section suivante discute la – preuve de – correction de l'approche. Finalement les deux dernières sections présentent les résultats et un bref état de l'art avant de conclure.

2. Quelques mots sur l'interprétation abstraite

2.1. *Théorie*

On se place dans le cadre de l'analyse statique par interprétation abstraite. Ce cadre théorique permet de *calculer* une sémantique abstraite, sur-approximation[1] de la véritable sémantique – généralement non calculable – d'un programme, permettant ainsi d'en déduire des propriétés si l'approximation est assez fine. La théorie initialement développée à la fin des années 1970 [CC92] a depuis connu des applications industrielles [DS07].

Cette section constitue une – brève – introduction des quelques principes de l'interprétation abstraite qui nous seront utiles par la suite.

Définition 2.1 *Un* treillis complet *est un ensemble A muni d'un ordre (relation réflexive, transitive et antisymétrique) partiel $\sqsubseteq$ et d'une borne supérieure $\bigsqcup$: pour toute partie S de A,*

$$\begin{array}{ll} \forall x \in S, x \sqsubseteq \bigsqcup S & (\bigsqcup S \text{ est un majorant de } S) \\ \forall y \in A, (\forall x \in S, x \sqsubseteq y) \Rightarrow \bigsqcup S \sqsubseteq y & \text{(et c'est le plus petit)} \end{array}$$

Propriété 2.1 *Un treillis complet est automatiquement muni d'une borne inférieure $\bigsqcap$ (plus grand minorant) $\bigsqcap S = \bigsqcup \{y \mid \forall x \in S, y \sqsubseteq x\}$ ainsi que d'un plus petit élément $\bot = \bigsqcup \emptyset = \bigsqcap S$ et d'un plus grand élément $\top = \bigsqcup S = \bigsqcap \emptyset$.*

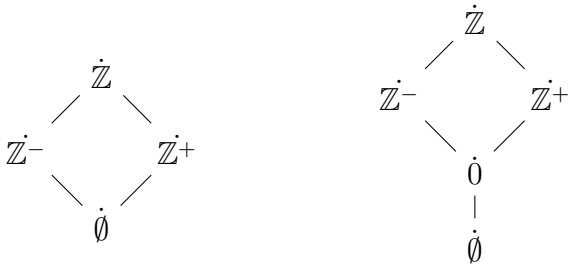
Exemple 2.1 *Pour tout ensemble (par exemple $\mathbb{Z}$), l'ensemble de ses parties ($\mathcal{P}(\mathbb{Z})$) muni de l'ordre inclusion $\subseteq$ et de la borne supérieure union $\cup$ est un treillis complet. Ce treillis a pour borne inférieure l'intersection $\cap$, pour plus petit élément l'ensemble vide $\emptyset$ et pour plus grand élément l'ensemble au complet ($\mathbb{Z}$).*

Les états possible du programme étant représentés par un treillis complet $(S, \sqsubseteq, \bigsqcup)$ (par exemple $\mathcal{P}(\mathbb{Z})$ pour les valeurs prises par une

1. On pourrait théoriquement aussi calculer une sous approximation, mais c'est en pratique moins utilisé.

variable entière) sont abstraits par un treillis $(S^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp)$. Le lien entre le concret S et l'abstrait $S^\sharp$ est donné par une fonction – dite de concrétisation – $\gamma : S^\sharp \to S$. On dit alors que l'état abstrait $s^\sharp \in S^\sharp$ est une sur-approximation de l'état concret $s \in S$ si $s \sqsubseteq \gamma(s^\sharp)$.

Exemple 2.2 *Deux exemples de treillis abstraits pour $\mathcal{P}(\mathbb{Z})$, l'ordre $\sqsubseteq$ correspondant à « être plus bas » sur le dessin :*



$\gamma : \dot{\mathbb{Z}} \mapsto \mathbb{Z}, \dot{\mathbb{Z}^-} \mapsto \{x \in \mathbb{Z} \mid x \leqslant 0\}, \dot{\mathbb{Z}^+} \mapsto \{x \in \mathbb{Z} \mid x \geqslant 0\},$
$\dot{0} \mapsto \{\, 0 \,\}, \dot{\emptyset} \mapsto \emptyset.$

Pour une opération élémentaire f du programme concret (par exemple l'addition $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$), $f^\sharp$ est une *abstraction correcte* de f si :

$$\forall x_1^\sharp, \ldots, x_n^\sharp \in S^\sharp, f\left(\gamma(x_1^\sharp), \ldots, \gamma(x_n^\sharp)\right) \sqsubseteq \gamma\left(f^\sharp\left(x_1^\sharp, \ldots, x_n^\sharp\right)\right)$$

Étant donné un treillis abstrait et une abstraction correcte de chaque opération élémentaire du langage, la théorie de l'interprétation abstraite permet d'obtenir une méthode de calcul effective d'une sur-approximation de la sémantique des programmes du langage.

Nous avons maintenant une définition d'abstraction correcte. Mais parmi ces abstractions correctes, certaines peuvent être plus précises que d'autres. On suppose par la suite que γ est monotone (pour tout $s^\sharp$ et $s'^\sharp$ de $S^\sharp$ tels que $s^\sharp \sqsubseteq^\sharp s'^\sharp$ alors $\gamma(s^\sharp) \sqsubseteq \gamma(s'^\sharp)$), ainsi l'ordre $\sqsubseteq^\sharp$ sur les abstractions peut se lire « est plus précis que », ce qui donne tout son sens à la définition suivante.

Définition 2.2 $s^\sharp \in S^\sharp$ *est la* meilleure abstraction *d'un élément concret* $s \in S$ *s'il est plus petit que toutes les autres sur-approximations de* s *:* $\forall s'^\sharp, s \sqsubseteq \gamma(s'^\sharp) \Rightarrow s^\sharp \sqsubseteq^\sharp s'^\sharp$.

Cette meilleure abstraction n'existe pas toujours, comme en témoigne le treillis de gauche de l'exemple 2.2 dans lequel $\dot{\mathbb{Z}}^-$ et $\dot{\mathbb{Z}}^+$ sont tous deux des sur-approximations de 0 mais sont incomparables.

Définition 2.3 *Soit* $(S, \sqsubseteq, \bigsqcup)$ *et* $(S^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp)$ *deux treillis complet. Une paire de fonctions* $\alpha : S \to S^\sharp$ *et* $\gamma : S^\sharp \to S$ *forme une* correspondance de Galois *si :*

$$\forall s \in S, \forall s^\sharp \in S^\sharp, \alpha(s) \sqsubseteq^\sharp s^\sharp \Leftrightarrow s \sqsubseteq \gamma(s^\sharp)$$

Théorème 2.1 *Étant donné deux treillis complet* S *et* $S^\sharp$ *et une fonction* $\gamma : S^\sharp \to S$*, il existe une meilleure abstraction dans* $S^\sharp$ *pour tout élément de* S *si et seulement s'il existe une fonction* $\alpha : S \to S^\sharp$ *formant une correspondance de Galois avec* γ.

Dans ce cas, la meilleure abstraction $f^\sharp$ d'une opération f dans S est alors donnée par : $f^\sharp = \alpha \circ f \circ \gamma$.

2.2. *Implantation*

On ne s'intéresse par la suite qu'aux domaines abstraits eux-mêmes, c'est à dire aux treillis $S^\sharp$ munis des opérations abstraites $f^\sharp$ et non au langage analysé ni au moteur de point fixe de l'analyseur.

Définir un domaine consiste alors à implanter un module OCaml qui satisfait les signatures suivantes. Il doit définir un ensemble de valeurs de type t muni d'une structure de treillis :

```
module type Lattice = sig
  type t
  val eq : t -> t -> bool  (* égalité *)
  val order : t -> t -> bool  (* ⊑♯ *)
```

```
  val join : t -> t -> t  (* x♯, y♯ ↦ ⨆ { x♯, y♯ } *)
  val meet : t -> t -> t  (* x♯, y♯ ↦ ⨅ { x♯, y♯ } *)
  val top : t  (* ⊤ *)
  val bottom : t  (* ⊥ *)
  val to_string : t -> string  (* fonction d'affichage *)
end
```

Et, dans le cas de l'abstraction de domaines numériques, il doit également définir les fonctions abstraites correspondant aux opérateurs concrets sur $\mathbb{Z}^2$:

```
module type IntAbsDom = sig
  include Lattice

  val add : t -> t -> t  (* addition en avant *)
  val sub : t -> t -> t  (* soustraction en avant *)
  val mul : t -> t -> t  (* multiplication en avant *)
  val add_bwd : t -> t -> t -> t * t  (* addition en arrière *)
  val sub_bwd : t -> t -> t -> t * t  (* soustraction en arrière *)
  val mul_bwd : t -> t -> t -> t * t  (* multiplication en arrière *)
end
```

Les opérations arrières prennent deux opérandes $x^\sharp$ et $y^\sharp$ et un résultat $z^\sharp$ et raffinent $x^\sharp$ et $y^\sharp$ sachant que leur résultat est surapproximé par $z^\sharp$.

Exemple 2.3 *En reprenant le treillis de droite de l'exemple 2.2,* $\mathrm{add\_bwd}\ \dot{\mathbb{Z}}^-\ \dot{\mathbb{Z}}^-\ \dot{0} = \dot{0}, \dot{0}$ *: si la somme de deux entiers négatifs est nulle alors ils sont tous deux nuls.*

3. Construction du treillis

L'objectif est d'automatiser le plus possible la génération de domaines abstraits finis – c'est à dire tels que $S^\sharp$ est un ensemble fini. Ces

2. On pourrait appliquer de manière identique la méthode présentée dans cet article à n'importe quelle théorie implantée dans un solveur SMT, mais on se concentrera ici sur les entiers $\mathbb{Z}$.

domaines sont généralement utilisés pour faire communiquer d'autres domaines entre eux, ou comme des domaines simples pour s'initier à l'interprétation abstraite.

Cette section présente la génération du code OCaml du treillis d'après le schéma général suivant :

– lecture de la description minimale de l'ordre partiel (sous forme de diagramme de Hasse) ;

– calcul de la clôture réflexive transitive (relation d'ordre) à partir du graphe précédent ;

– calcul pour chaque paire des bornes supérieures et inférieures ;

– calcul des extremums $\top$ et $\bot$.

En effet, on peut noter que dans le cas de treillis finis, la donnée d'une borne supérieure $\bigsqcup^\sharp : \mathcal{P}(S^\sharp) \to S^\sharp$ est équivalente à la donnée d'un plus petit élément $\bot$ et d'une borne supérieure binaire $\sqcup^\sharp : S^\sharp \times S^\sharp \to S^\sharp$.

3.1. *Interpréter le diagramme de Hasse*

3.1.1. *Principe*

Définition 3.1 *Un* diagramme de Hasse *est une représentation graphique, similaire au dessin d'un graphe, d'un ordre partiel $\sqsubseteq$ sur un ensemble fini S tel que :*

– les éléments de l'ensemble sont représentés par des points ;

– si un élément $x \in S$ est inférieur – selon l'ordre $\sqsubseteq$ – à un élément $y \in S$, on dessine le point x plus bas que y ;

– on trace un trait entre les point x et y si et seulement si x est directement inférieur à y ($x \sqsubseteq y$ et pour tout z, si $x \sqsubseteq z \sqsubseteq y$ alors $z = x$ ou $z = y$).

Le graphe représenté est donc – d'après le dernier point – la *réduction réflexive transitive* de l'ordre partiel du treillis. Cela permet d'éviter de charger le dessin avec des arrêtes implicites puisqu'un ordre est nécessairement réflexif et transitif. On notera également qu'il est inutile d'orienter les arrêtes par des flèches puisque – de par le deuxième point – si deux éléments sont reliés par une arrête, celui dessiné le plus bas

est nécessairement le plus petit. L'ordre étant antisymétrique, le graphe est acyclique et une telle disposition est toujours possible.

Exemple 3.1 *Les deux dessins de l'exemple 2.2 sont des diagrammes de Hasse.*

En pratique, nous avons choisi deux formats d'entrée : un format représentant le diagramme de Hasse sous forme d'ASCII art et l'autre dans le langage de description de graphes DOT.

3.1.2. *Implantation : format ASCII*

Camlp4 fourni un mécanisme de *quotations* : tout le texte compris entre <:type_quotation< et >> ne sera pas interprété comme du code OCaml mais passé comme une chaîne de caractères à une fonction définie par l'utilisateur pour `type_quotation` et remplacé dans le code source par le code OCaml généré par cette fonction.

Exemple 3.2 *Nous allons compiler le diagramme de Hasse représenté par le code Camlp4 ci dessous à gauche en le motif de filtrage OCaml à droite :*

```
<:finite_lattice<
     STop
    /    \
SLe0      SGe0
    \    /
     SZero
       |
       |
     SBot
>>
```

```
SBot, SZero
| SZero, SLe0
| SZero, SGe0
| SLe0, STop
| SGe0, STop
```

Grâce à la disposition de bas en haut des diagrammes de Hasse, la lecture de l'ASCII art s'avère plus aisée qu'on aurait pu le craindre.

```
(* on va utiliser Camlp4 pour générer le résultat *)
```

```
open Camlp4.PreCast.Syntax
(* ensemble de sommets *)
module VSet = Set.Make
  (struct type t = string let compare = compare end)

let parse _loc _ s =  (* s : chaîne de caractères de la quotation *)
```

On lit la chaîne de caractères ligne par ligne en utilisant essentiellement trois tables de hachage :

```
  let prev = ref (Hashtbl.create 7) in
```

Associe à chaque point (numéro de colonne dans la ligne) de la ligne précédente l'ensemble des sommets du graphe auquel il est lié.

```
  let curr = ref (Hashtbl.create 7) in
```

Idem mais pour la ligne en cours de lecture.

```
  let edges = (Hashtbl.create 7) in
```

Arrêtes du graphe déjà lues.

```
  let add_to_pos i ids =
    let ids' = try Hashtbl.find !curr i with Not_found -> VSet.empty in
    let ids = VSet.union ids ids' in
    Hashtbl.add !curr i ids in
```

Ajoute l'ensemble de sommets `ids` à la position `i` dans la table `curr`.

```
  let new_line () =
    let swap a b = let t = !a in a := !b ; b := t in
    Hashtbl.clear !prev ;
    swap prev curr in
```

On en a fini avec la table `prev` et `curr` devient `prev`.

```
  let parse_id b id =  (* b : position du début de l'identifiant id *)
    let e = b + String.length id in  (* position de la fin de id *)
    let above = ref VSet.empty in
```

```
  for i = b to e do
    begin try above := VSet.union (Hashtbl.find !prev i) !above
    with Not_found -> () end ;
    add_to_pos i (VSet.singleton id) ;
  done ;
  Hashtbl.add edges id !above ; in
```

Le nouveau sommet `id` est lié – dans la table `edges` – à tous ceux auxquels sont liés les points juste au dessus de lui – dans la table `prev`. Et les points juste en dessous sont liés au nouveau sommet – dans la table `curr` par `add_to_pos`.

```
  let link to_i from_i = add_to_pos from_i (Hashtbl.find !prev to_i) in
```

Crée un lien entre `from_i` et `to_i`. Il ne reste maintenant plus qu'à utiliser les fonctions que l'on vient de définir. `Lexer` est un analyseur lexical – produit par OCamlLex – produisant les lexèmes apparaissant dans le code suivant en gardant trace de leur position dans la ligne lue (paramètre `c`).

```
  let lexbuf = Lexing.from_string s in
  begin try while true do match Lexer.token lexbuf with
    | Tokens.NEWLINE -> new_line ()
    | Tokens.VERTEX(c, s) -> parse_id c s
    | Tokens.BACKSLASH c -> link c (c + 1)
    | Tokens.PIPE c -> link c c
    | Tokens.SLASH c -> link (c + 1) c
    | Tokens.X c -> link c (c + 1) ; link (c + 1) c
  done with Failure "lexing: empty token" -> () end ;
```

On génère finalement le résultat sous forme d'un motif de filtrage grâce à Camlp4.

```
  Hashtbl.fold
    (fun v -> VSet.fold (fun v' patt ->
      < :patt< $patt$ | $uid :v$, $uid :v'$ >>))
    edges < :patt< >>
```

Et on enregistre la quotation au près de Camlp4.

```
let _ = Quotation.add
  "finite_lattice" Quotation.DynAst.patt_tag parse ;
```

Le parseur que l'on vient de présenter ne contient aucune gestion d'erreur. Il est relativement facile de vérifier les propriétés suivantes :

– tous les sommets ont des noms différents ;

– il n'y a pas d'arête menant nulle part ;

– il n'y a pas d'arête superflue (découlant de la transitivité).

En revanche le code devient sensiblement plus volumineux (une centaine de lignes).

On notera qu'on se trouve ici dans l'une des quelques situations dans lesquelles on peut tirer parti des fonctionnalités de programmation impérative du langage OCaml.

3.1.3. *Implantation : format DOT*

Bien qu'il soit en théorie possible de dessiner n'importe quel treillis dans le format ASCII art précédent, cela peut parfois s'avérer malcommode – en particulier pour des treillis assez larges – d'où l'intérêt d'un format d'entrée alternatif. En pratique, on enregistre une quotation `dot_lattice` auprès de Camlp4 usant du parseur DOT d'Ocamlgraph.

3.2. *Ordre partiel et opérations primitives du treillis*

3.2.1. *Principes*

Ainsi qu'on l'a noté dans l'introduction de la présente section, après avoir lu le diagramme de Hasse, on obtient la relation d'ordre $\sqsubseteq^\sharp$ par une opération de clôture réflexive transitive. Il reste alors à vérifier que l'on est bien en présence d'un treillis en s'assurant de l'existence d'un élément minimum $\bot$ et d'une borne supérieure binaire $\sqcup^\sharp$.

3.2.2. *Implantation*

Pour les manipulations d'ensembles partiellement ordonnés, on utilise un module `Poset` basé sur Ocamlgraph [CFS07] auquel sont ajouté

les opérations `min`, `max`, `lub` (borne supérieure) et `glb` (borne inférieure).

3.2.2.1. Ordre

On utilise le graphe g de la clôture réflexive transitive de l'ordre.

```
let generate_order _loc g =
 let patt = Poset.fold_edges
  (fun v -> VSet.fold
   (fun v' patt -> < :patt< $patt$ | $uid :v$, $uid :v'$ >>))
  g < :patt< >> in
 < :expr< fun a b -> match a, b with $patt$ -> true | _ -> false >>
```

3.2.2.2. Bornes supérieure et inférieure

Elles sont calculées pour chaque paire de valeurs, ce qui permet d'en vérifier l'existence. Le code de génération est relativement similaire à `generate_order`.

3.2.2.3. Extremums

$\top$ et $\bot$ sont calculées respectivement comme `Poset.glb g VSet.empty` et `Poset.lub g VSet.empty`, vérifiant ainsi leur existence.

4. Synthétiser la sémantique : la fonction de concrétisation

Une fois le treillis construit, nous nous intéressons à la construction automatique, autant que faire se peut, des opérations de transfert sémantique : les opérations arithmétiques addition, soustraction et multiplication dans notre cas. L'approche que nous avons choisie consiste à utiliser des solveurs SMT et une description du sens des valeurs du treillis abstraits, c'est à dire une fonction de concrétisation.

4.1. *Spécification*

Ayant donnée le treillis $S^\sharp$, l'utilisateur doit encore préciser une fonction de concrétisation $\gamma : S^\sharp \to S$ (S étant supposé être l'ensemble

$\mathcal{P}(\mathbb{Z})$ des parties de $\mathbb{Z}$)[3]. On utilise pour cela la grammaire Camlp4 suivante :

```
EXTEND Gram
GLOBAL : str_item ;
str_item :
 [ [ "abstract_domain" ; id = UIDENT ;
     "on" ; order = patt ;
     "with" ; concr = gamma -> generate _loc id order concr ] ] ;
gamma :
 [ [ "gamma" ; v = LIDENT ; "=" ; "function" ; OPT "|" ;
   b = gamma_body -> _loc, v, b ] ] ;
gamma_body :
 [ [ c = gamma_case ; "|" ; l = SELF -> c : :l
   | c = gamma_case -> [c] ] ] ;
gamma_case :
 [ [ c = UIDENT ; "->" ; s = STRING -> _loc, c, s ] ] ;
END
```

dans laquelle pour chaque élément `c` de $S^\sharp$ les chaînes de caractère `s` représentent `v` $\in \gamma($`c`$)$ sous la forme d'une formule dans le langage Smtlib [BST10], utilisé pour les compétitions de solveurs SMT.

Exemple 4.1 *Spécification complète du domaine abstrait des exemples précédents :*

```
abstract_domain Sign on
<:finite_lattice<
     STop
    /    \
SLe0      SGe0
    \    /
```

3. On pourrait en fait se contenter de la fonction de concrétisation et en déduire le treillis puisque, en présence d'une correspondance de Galois, l'ordre $\sqsubseteq^\sharp$ est défini par : $x^\sharp \sqsubseteq^\sharp y^\sharp \Leftrightarrow \gamma(x^\sharp) \sqsubseteq \gamma(y^\sharp)$. Mais il est parfois préférable – par exemple pour des applications dans l'enseignement – de donner les deux. De plus, cette redondance laisse plus de chance de découvrir une éventuelle erreur dans la définition de la fonction de concrétisation.

```
      SZero
       |
       |
      SBot
>>
with gamma x = function
  | STop -> "true"
  | SLe0 -> "(<= x 0)"
  | SGe0 -> "(>= x 0)"
  | SZero -> "(= x 0)"
  | SBot -> "false"
```

4.2. *Correction*

Ainsi que nous l'avons vu dans la section 2.1, la fonction γ doit vérifier certaines propriétés afin d'assurer l'existence d'une – meilleure – abstraction pour les opérations de transfert sémantique :

– monotonie de γ : pour tout $x^\sharp$ et $y^\sharp$ de $S^\sharp$ si $x^\sharp \sqsubseteq^\sharp y^\sharp$ alors $\gamma(x^\sharp) \sqsubseteq \gamma(y^\sharp)$, [4] ;

– $\gamma(\top) = \mathbb{Z}$: garantissant qu'il existe toujours une abstraction correcte ;

– correction de la borne inférieure binaire : pour tout $x^\sharp$ et $y^\sharp$ de $S^\sharp$, $\gamma(x^\sharp) \sqcap \gamma(y^\sharp) \sqsubseteq \gamma\left(x^\sharp \sqcap^\sharp y^\sharp\right)$[5].

Un domaine abstrait ne vérifiant pas la première condition n'ayant guère de sens, on générera une erreur si elle n'est pas vérifiée. Il en sera de même pour la seconde condition. En revanche, la troisième en conjonction avec les deux précédentes revient à dire que γ forme une correspondance de Galois avec $\alpha : x \mapsto \bigsqcap \left\{x^\sharp \;\middle|\; x \sqsubseteq \gamma(x^\sharp)\right\}$, garantissant ainsi l'existence d'une meilleure abstraction pour toute partie de $\mathbb{Z}$. On se contentera donc d'un avertissement signalant qu'il pourrait ne pas

4. Revient à dire que la borne supérieure $\bigsqcup^\sharp$ est une abstraction correcte de $\bigsqcup$: pour tout partie $A^\sharp$ de $S^\sharp$, $\bigsqcup \gamma(A^\sharp) \sqsubseteq \gamma\left(\bigsqcup^\sharp A^\sharp\right)$.

5. En conjonction avec le point précédent cela est équivalent à la correction de la borne inférieure – ensembliste.

exister de meilleure abstraction – pour les fonction calculées – si elle est enfreinte.

En pratique, pour vérifier ces propriétés, on dispose d'une fonction `Smt.ask` prenant une formule Smtlib et retournant `true` ou `false` si un solveur SMT prouve la formule respectivement satisfaisable ou insatisfaisable. Une exception `Smt.No_answer` est levée si aucun solveur ne parvient à conclure dans un délai raisonnable. On vérifiera ainsi respectivement la non satifaisabilité des formules suivantes :

– $x \in \gamma(x^\sharp) \wedge \neg \left(y \in \gamma(y^\sharp)\right)$ pour tout $x^\sharp$ et $y^\sharp$ de $S^\sharp$ tels que $x^\sharp$ est directement inférieur à $y^\sharp$;

– $\neg \left(x \in \gamma(\top)\right)$;

– $z \in \gamma(x^\sharp) \wedge z \in \gamma(y^\sharp) \wedge \neg \left(z \in \gamma\left(x^\sharp \sqcap^\sharp y^\sharp\right)\right)$ pour tout $x^\sharp$ et $y^\sharp$ de $S^\sharp$ incomparables entre eux.

5. Sémantique avant

5.1. *Principe*

On va calculer pour chaque paire $(x^\sharp, y^\sharp)$ d'éléments de $S^\sharp$ la meilleure abstraction – si elle existe – pour les opérations arithmétiques $+$, $-$ et $\times$. On rappelle qu'un opérateur binaire $binop^\sharp$ sur $S^\sharp$ est une abstraction correcte d'un opérateur concret $binop$ sur S s'il vérifie :

$$\forall x^\sharp, y^\sharp \in S^\sharp, \forall x, y, z \in S, x \in \gamma(x^\sharp) \wedge y \in \gamma(y^\sharp) \wedge z = x \; binop \; y \Rightarrow z \in \gamma\left(x^\sharp \; binop^\sharp \; y^\sharp\right)$$

d'où une méthode très simple pour calculer la meilleure abstraction (ou au moins une bonne abstraction s'il n'en existe pas de meilleure) d'un opérateur binaire $binop$ pour $x^\sharp$ et $y^\sharp$ donnés :

– **séléction des candidats corrects :** pour tout $z^\sharp \in S^\sharp$, demander aux solveurs SMT si $z^\sharp$ est une sur-approximation de $\gamma(x^\sharp) \; binop \; \gamma(y^\sharp)$, si aucun solveur n'est capable de répondre il est correct – quoique potentiellement imprécis – de considérer le résultat comme négatif ;

– **choix du plus petit :** prendre le plus petit des $z^\sharp$ ayant passé le test précédent, si l'ensemble n'a pas de minimum (pas de meilleure abstraction), on prend un élément minimal quelconque (il y en a plu-

sieurs, incomparables entre eux) et on avertit l'utilisateur de l'absence de meilleure abstraction.

On notera qu'on a toujours existence d'une abstraction, $\top$ étant toujours une sur-approximation correcte du fait de l'hypothèse $\gamma(\top) = \mathbb{Z}$.

5.2. *Implantation*

On dispose d'une fonction `in_gamma` qui prenant une fonction de concrétisation γ – telle que lue à la section précédente – un élément $x^\sharp$ et une variable x retourne une formule Smtlib représentant $x \in \gamma(x^\sharp)$. On teste ainsi la non satisfaisabilité des formules $x \in \gamma(x^\sharp) \wedge y \in \gamma(y^\sharp) \wedge z = x \; binop \; y \wedge \neg \left(z \in \gamma(z^\sharp)\right)$ pour $x^\sharp$, $y^\sharp$ et $z^\sharp$ donnés.

```
let test_fwd gamma binop x y z =
 let in_gamma_x = in_gamma gamma x "x" in
 let in_gamma_y = in_gamma gamma y "y" in
 let in_gamma_z = in_gamma gamma z "z" in
 let formula = "(and\n" ^
  in_gamma_x ^ "\n" ^
  in_gamma_y ^ "\n" ^
  "(= z (" ^ binop ^ " x y))\n" ^
  "(not " ^ in_gamma_z ^ "))\n" in
 try not (Smt.ask formula)
 with Smt.No_answer -> false

let determine_fwd _loc g gamma binop x y =
 let upper_bounds = Poset.fold_vertex
  (fun candidate candidates ->
    if test_fwd gamma binop x y candidate then
     VSet.add candidate candidates
    else
     candidates)
  g VSet.empty in
 try
  Poset.min g upper_bounds
 with Poset.No_min_max ->
```

```
warning _loc ("No best abstraction for " ^ x ^ " " ^ binop ^
              " " ^ y ^ ".");
Poset.choose_minimal g upper_bounds
```

5.3. *Optimisations*

L'opération la plus coûteuse est l'appel aux solveurs SMT (typiquement plusieurs centièmes de secondes par appel). Il est donc intéressant d'en limiter le nombre grâce aux optimisations suivantes :

– la meilleure abstraction $binop^\sharp$ étant monotone (pour tout $x^\sharp$, $y^\sharp$, $x'^\sharp$ et $y'^\sharp$, si $x^\sharp \sqsubseteq^\sharp x'^\sharp$ et $y^\sharp \sqsubseteq y'^\sharp$ alors $x^\sharp\ binop^\sharp\ y^\sharp \sqsubseteq^\sharp x'^\sharp\ binop^\sharp\ y'^\sharp$) on peut se contenter de ne tester que les éléments supérieurs aux valeurs déjà calculées avec des opérandes plus petit ;

– lorsqu'un solveur prouve qu'un élément est un candidat correct il suffit de continuer à chercher parmi les éléments qui lui sont inférieurs ;

– au contraire lorsqu'un élément n'est pas un candidat valide, il est inutile de tester les éléments plus petits.

L'ordre dans lequel sont testés les éléments du treillis influe donc sur le nombre d'appels aux solveurs. En pratique, commencer par ceux ayant le plus de voisins dans le graphe du diagramme de Hasse semble une heuristique raisonnable. Pour exploiter la monotonicité, il est par contre optimal d'effectuer les calculs pour les paires $(x^\sharp, y^\sharp)$ dans un ordre croissant.

6. Sémantique arrière

6.1. *Principe*

Malgré quelques différences, il y a de fortes similarités avec la sémantique avant. On va calculer pour chaque triplet $(x^\sharp, y^\sharp, z^\sharp)$ d'éléments de $S^\sharp$ le meilleur raffinement $(x'^\sharp, y'^\sharp)$ – s'il existe – de $x^\sharp$ et $y^\sharp$ pour les opérations arithmétiques $+$, $-$ et $\times$. Un opérateur arrière

$binop^\sharp\!\!\downarrow : {S^\sharp}^3 \to {S^\sharp}^2$ est correct par rapport à l'opérateur concret $binop$ s'il vérifie :

$$\begin{aligned}\forall x^\sharp, y^\sharp, z^\sharp, x'^\sharp, y'^\sharp \in S^\sharp, \forall x, y, z \in S, x \in \gamma(x^\sharp) \wedge y \in \gamma(y^\sharp) \wedge \\ z = x \; binop \; y \wedge z \in \gamma(z^\sharp) \wedge \\ (x'^\sharp, y'^\sharp) = binop^\sharp\!\!\downarrow(x^\sharp, y^\sharp, z^\sharp) \Rightarrow \\ x \in \gamma(x'^\sharp) \wedge y \in \gamma(y'^\sharp)\end{aligned}$$

On remarque qu'on peut calculer indépendamment les deux composantes $x'^\sharp$ et $y'^\sharp$ du résultat, ce qui permet dans les deux cas d'utiliser une méthode similaire à celle choisie pour la sémantique avant. On notera également qu'on a toujours existence d'une solution, le couple $(x^\sharp, y^\sharp)$ donné en entrée étant une solution correcte, il suffit donc de tester les éléments inférieurs.

6.2. *Implantation*

Pour $x^\sharp$, $y^\sharp$ et $z^\sharp$ donnés, on calcule indépendamment les deux composantes du résultat. Pour la première, on teste pour tout $x'^\sharp \sqsubseteq x^\sharp$ la non satisfaisabilité de la formule $x \in \gamma(x^\sharp) \wedge y \in \gamma(y^\sharp) \wedge z = x \; binop \; y \wedge z \in \gamma(z^\sharp) \wedge \neg \left(x \in \gamma(x'^\sharp)\right)$ et on garde la plus petite solution. On procède de même pour la deuxième composante $y'^\sharp$.

6.3. *Optimisations*

On peut appliquer les mêmes optimisations que pour la sémantique avant.

7. Nettoyage

On factorise les motifs de filtrage suivant le résultat. Cela ne modifie évidemment en rien l'efficacité du code une fois compilé mais permet d'obtenir un résultat un peu plus lisible – si tant est qu'il puisse l'être – ce qui est indispensable pour réaliser des exercices à trous dans une optique d'enseignement de l'interprétation abstraite par exemple.

8. Application : le treillis des signes étendus

La description suivante :

```
abstract_domain ESign on
<:finite_lattice<
     STop
    /  | \
   /   |  \
  /    |   \
SLe0  SN0 SGe0
 | \ /  \ / |
 |  X    X  |
 | / \  / \ |
SLt0  S0  SGt0
  \    |   /
   \   |  /
    \  | /
     SBot
>>
with gamma x = function
  | STop -> "true"
  | SLe0 -> "(<= x 0)"
  | SN0 -> "(not (= x 0))"
  | SGe0 -> "(>= x 0)"
  | SLt0 -> "(< x 0)"
  | S0 -> "(= x 0)"
  | SGt0 -> "(> x 0)"
  | SBot -> "false"
```

sera compilée en (extrait des plus de 500 lignes de code OCaml générées) :

type eSign_base_t = | STop | SN0 | SLt0 | SLe0 | SGt0 | SGe0 | SBot | S0

module ESignLat : LatticeDef.Lattice **with type** t = eSign_base_t = **struct**
  type t = eSign_base_t

```
  let eq = ( = )
  let order a b = match (a, b) with
    | (S0, S0) | (S0, SGe0) | (S0, SLe0) | (S0, STop) | (SBot, S0) [...]
    | (SN0, STop) | (STop, STop) -> true
    | _ -> false
  let join a b = match (a, b) with
    | (SBot, S0) | (S0, SBot) | (S0, S0) -> S0 | (SGt0, SGe0)
    | (SGt0, S0) | (SGe0, SGt0) | (SGe0, SGe0) | (SGe0, SBot)
    | (SGe0, S0) | (SBot, SGe0) | (S0, SGt0) | (S0, SGe0) -> SGe0
    | (SLt0, SLe0) | (SLt0, S0) | (SLe0, SLt0) | (SLe0, SLe0)
    | (SLe0, SBot) | (SLe0, S0) [...]
  let meet a b = [...]
  let top = STop
  let bottom = SBot

  let to_string = function
    | S0 -> "S0"
    | SBot -> "SBot" [...]

  let sem_add a b = match (a, b) with
    | (S0, S0) -> S0
    | (STop, SBot) | (SN0, SBot) | (SLt0, SBot) | (SLe0, SBot) [...]
    | (SBot, SBot) | (SBot, S0) | (S0, SBot) -> SBot [...]
  let sem_sub a b = [...]
  let sem_mul a b = [...]

  let sem_add_bwd a b c = match (a, b, c) with
    | (STop, SBot, STop) | (SN0, SBot, STop) | (SLt0, SBot, STop) [...]
    | (S0, SGe0, S0) -> (S0, S0) [...]
    | _ -> (a, b)
  let sem_sub_bwd a b c = [...]
  let sem_mul_bwd a b c = [...]
end
```

9. Vérification du code généré

La correction du code généré dépend de principalement deux facteurs : la correction de notre code (un millier de lignes de CaML) et la correction des solveurs SMT.

Une solution serait de générer en lieu et place du code OCaml un code pour un assistant de preuve comme Coq dont on extrairait le code exécutable. Des travaux sur la réalisation d'interprète abstraits à l'aide de Coq existent [Pic05] mais s'il est très facile de générer un treillis vérifiant ces signatures de modules (les scripts de preuve pour la correction des bornes supérieure et inférieure étant constants dans le cas fini) les fonctions sémantiques posent un problème autrement sérieux (l'usage de tactiques très automatisées telles `psatz` semble toutefois une solution extraordinairement efficace). L'idéal serait de disposer de preuves données par les solveurs SMT de leurs réponses ou de prouveurs SMT corrects par construction. Des travaux existent dans cette direction, mais c'est encore un sujet de recherche actif [CCKL07].

Une solution bon marché pour tenter d'augmenter la confiance en les prouveurs pourrait être d'exiger un résultat concordant de plusieurs d'entre eux.

10. Résultats

On obtient un code d'une taille modérée : guère plus d'un millier de lignes de CaML[6]. Le tableau 1 donne le nombre d'appels aux solveurs SMT et les temps d'exécution (sur un Core 2 @ 1.20 GHz) pour quelques exemples classiques. La complexité étant un $\Omega(n^3)$ pour un treillis de taille n, on est en pratique limité à des domaines abstraits d'au plus quelque dizaines d'éléments. On peut d'autre part noter que la meilleure abstraction est obtenue pour tous les exemples, à l'exception de `parity` et `mod3` pour lesquels on doit fournir manuellement une négation de la fonction de concrétisation pour obtenir des résultats décents (i.e. meilleure abstraction à l'exception de la multiplication en

6. Disponible à : `http://cavale.gforge.enseeiht.fr/files/dessine_moi_un_domaine_abstrait_fini.tgz`.

Domaine	taille de $S^\sharp$	sans optimisation appels (temps total)	avec optimisations appels (temps total)
bot_top	2	49 (1 s)	40 (1 s)
eq_zero	4	628 (6 s)	339 (4 s)
signs	4	628 (7 s)	300 (3 s)
signs_with_zero	5	1540 (17 s)	594 (7 s)
extended_signs	8	7500 (83 s)	2429 (33 s)
parity	4	617 (8 s)	325 (4 s)
mod3	5	1294 (147 s)	600 (117 s)

Tableau 1 – Nombre d'appels aux solveurs SMT pour différents exemples (les domaines `signs` et `signs_with_zero` correspondent à ceux de l'exemple 2.2 respectivement à gauche et à droite et `extended_signs` à celui de la section 8)

arrière[7]). On note enfin que la multiplication (opération non linéaire) n'est pas très bien gérée par les solveurs SMT (à l'exception notable de Z3 qui parvient à donner certains résultats).

11. État de l'art et conclusion

On trouve mention de l'idée fondatrice de cet article dans la conclusion de [Pic08] mais nous n'avons pas connaissance d'une implémentation. D'autre part, d'autres travaux « plus sérieux » s'intéressent à la construction automatique de fonctions de transfert abstraites pour des blocs de code plutôt que pour de simples opérations[8] et à des domaines abstraits « infinis », par exemple le domaine abstrait des intervalles sur les réels voire même les flottants grâce à des techniques d'élimination de quantificateurs [Mon09] ou les domaine des intervalles [BK10] ou des congruences [KS08] sur les entiers machines à l'aide de décimation booléenne et de solveurs SAT. Le calcul de transformeurs abstraits à l'aide de solveurs a également été utilisé dans le cadre de l'analyse de structures en mémoire [YRS04].

7. Exemple d'obligation qu'aucun prouveur n'est capable de prouver non satisfaisable : $x = 3 * k\_x + 0 \wedge y = 3 * k\_y + 0 \wedge z = x * y \wedge z = 3 * k\_z + 1$.
8. En effet, la composition d'opérateurs abstraits élémentaires précis n'est généralement pas la meilleure abstraction de la composition des opérateurs concrets.

Nous avons présenté un outil simple de génération automatique de domaines abstraits finis permettant de remplacer l'écriture extrêmement fastidieuse de centaines de lignes de code par des descriptions graphiques de quelque lignes. Cet outil devrait trouver un premier usage dans l'enseignement de l'interprétation abstraite dispensé dans des écoles d'ingénieur : ISAE (ex. SUPAERO) et ENSEEIHT. Il sera ainsi possible de considérer des domaines abstraits comme les signes étendus, qui ne l'étaient pas auparavant.

Remerciements Nous tenons à remercier les relecteurs pour leurs remarques constructives.

Références

[BK10] Jörg Brauer and Andy King. Automatic Abstraction for Intervals Using Boolean Formulae. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2010.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org, 2010.

[CC92] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4) :511–547, 1992.

[CCKL07] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In *Second Automated Formal Methods workshop series (AFM07)*, Atlanta, Georgia, USA, November 2007.

[CCKL08] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X) : Semantic Combination of Congruence Closure with Solvable Theories. *Electronic Notes in Theoretical Computer Science*, 198(2) :51–69, May 2008.

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a Generic Graph Library using ML Functors. In *The Eighth Symposium on Trends in Functional Programming*, volume TR-SHU-CS-2007-04-1, pages XII/1–13, New York, USA, April 2007. Seton Hall University.

[Cp4] Camlp4. http://brion.inria.fr/gallium/index.php/Camlp4.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DS07] David Delmas and Jean Souyris. Astrée : From Research to Industry. In Hanne Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[KS08] Andy King and Harald Søndergaard. Inferring Congruence Equations Using SAT. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 281–293. Springer, 2008.

[Mon09] David Monniaux. Automatic modular abstractions for linear constraints. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 140–151. ACM, 2009.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories : From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). *J. ACM*, 53(6) :937–977, 2006.

[OC] OCaml. http://caml.inria.fr.

[Pic05] David Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french.

[Pic08] David Pichardie. Building Certified Static Analysers by Modular Construction of Well-founded Lattices. *Electronic Notes in Theoretical Computer Science*, 212 :225 – 239, 2008. Proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS 2008).

[YRS04] Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 530–545. Springer, 2004.