Action Concertée Incitative

SÉCURITÉ & INFORMATIQUE

APRON: Analyse de PROgrammes Numériques

Compte-rendu de la réunion du 16 décembre 2004: Interface commune

1 Objectifs et contexte

La réunion du 16 décembre avait pour but de lancer l'effort commun de définition d'une interface commune pour domaines abstraits. François Irigoin a tout d'abord présenté ses besoins dans le cadre de PIPS, puis Sebastian Pop a expliqué comment ses travaux dans gcc pouvaient s'appuyer sur une interface de domaines abstrait et Bertrand Jeannet a enchainé en proposant une interface fonctionnelle de niveau "treillis numérique", enfin, Duong Nguyen a fait le bilan des problèmes opérationnels à résoudre. L'après-midi a été consacrée à essayer de définir un opérateur a priori simple, *is_bottom*.

Ce compte-rendu comporte d'abord la liste des décisions prises d'un commun accord, des commentaires sur la liste d'opérateurs proposés par Bertrand Jeannet, une liste de problèmes restant à traiter, et une étude de cas, celui du prédicat *is_bottom*. Il ne rend pas compte de la problématique soulevée par Sebastian Pop. Elle devra être réexaminée en fonction des résultats de cette première journée de réunion pour vérifier que les besoins de Sebastian s'expriment bien en termes d'extracteurs.

Les transparents des quatre présentations sont en libre accès sur le site WEB du projet, `http://www.cri.ensmp.fr/apron`. Les participants étaient Nicolas Halbwachs, François Irigoin, Bertrand Jeannet, Antoine Miné (matin seulement), Duong Nguyen et Sebastian Pop.

2 Décisions

Les décisions prises ne sont pas classées par nature, mais simplement collectées chronologiquement.

1. Dans un premier temps, la problématique CRI de la gestion des liaisons entre dimensions, expressions d'adresse, *abstract locations* et identificateurs est laissée de côté pour commencer par quelque chose de plus simple. On se contente de manipuler un sous-ensemble dans un produit cartésien et on se restreint même à la puissance n d'un ensemble X, X^n.

2. La gestion des produits de domaines est laissée de côté, bien que le nommage générique puisse compliquer le problème ultérieurement.

3. La gestion du *mapping* et celle du *packing* sont laissées de côté au niveau 1 de l'interface.

4. Le type numérique entier de base utilisé dans l'analyseur est défini à la compilation et à l'édition de lien (comme dans PIPS, type *Value*). Il n'est pas prévu d'adapter dynamiquement ce type en fonction des pertes/des besoins de précision, e.g. passage en gmp.

5. Les signatures fonctionnelles et impératives seront toutes les deux supportées. Les objets seront recyclés ou non, bien que le recyclage ne soit pas utile pour les implémentations denses. Il n'a pas été décidé d'utiliser le passage d'un paramètre supplémentaire ou bien des noms de fonctions différents pour distinguer entre ces deux appels.

6. Les implémentations partielles seront acceptées, mais elles devront offrir toutes les signatures prévues pour permettre l'édition de lien et l'échec en cas d'appel à une fonction non implantée.

7. Vu l'évolution prévisible des architectures des ordinateurs, l'interface permettra d'écrire des implantations compatibles avec les threads, *threadsafe*.

8. Les débordements en capacité, en temps et en espace sont détectés, ou tout au moins, les mécanismes nécessaires à leur paramétrisation et à la récupération des incidents seront définis.

9. On définit au moins une version C de l'interface.

10. Pour certains opérateurs ou même pour chaque opérateur abstrait, il est possible de sélectionner une implémentation particulière, l'implémentation par défaut, l'implémentation donnant le résultat le plus précis ou l'implémentation donnant un résultat le plus rapidement possible. La perte de généricité correspondante semble moins gênante que l'absence de cette flexibilité, au moins pour certains opérateurs particulièrement sensibles. Les trois implémentations de base peuvent être identiques: l'association numéro vers algorithme peut être surjective.

11. Le retour d'un résultat mathématiquement exact peut être identifié à la demande de l'utilisateur (projection entière, enveloppe convexe, test de satisfiabilité en entier,...).

12. La factorisation, telle qu'elle est proposée par Grenoble, n'est pas visible à travers l'interface.

13. Le fait de garder ou non deux représentations duales, e.g. de polydèdres, n'est pas visible à travers l'interface.

14. Une notion de taille abstraite d'objet est définie.

15. Un contexte d'appel, *manager*, est passé explicitement en argument. Il sert à assurer la compatibilité avec les package de *threads* et à passer toutes les informations opérationnelles, e.g. dépassement en magnitude.

16. Le contenu du context d'appel n'est pas visible directement. Des primitives de construction et d'observation seront définies et fournies. On n'a pas à gérer une pile de contextes implicite. On n'a pas de variables globales. Le contexte n'est pas intégré aux objets du domaine abstrait.

17. Gestion mémoire: on ne supporte pas de comptage de références.

18. Proposer un système style *varags* plutôt que des listes ou des tableaux d'arguments quand le nombre d'arguments est variable.

19. Gestion des exceptions: l'interface n'expose aucune exception. Les exceptions internes doivent être traitées par l'implantation et l'information correspondant être masquée ou retournée via la structure *manager*.

20. Il n'est pas possible de définir certaines dimensions comme entières et d'autres comme rationnelles (demande de Nicolas Halbwachs). Le typage est global (cf. point sur X^n).

3 Liste des opérateurs

La liste des opérateurs proposés par Bertrand Jeannet est donnée en annexe. Cette section concerne essentiellement les modifications demandées.

Afin d'assurer performance des implantations et confort de l'utilisateur, l'objectif n'est pas d'avoir une liste d'opérateurs minimales, mais une liste d'opérateurs comportant les combinaisons d'opérateurs de base pouvant être simplifiées ou fréquemment utilisées par un analyseur. Les problèmes de mise au point, e.g. nommage symbolique des dimensions, et de validation, e.g. forme canonique, doivent aussi être pris en compte.

Par exemple, on proposera un système style *varargs* pour permettre d'enchainer les enveloppes convexes, les projections,... Chaque opérateur pouvant être étendu à la manière de *sum* et *prod* le sera. Cela a aussi un impact sur la fonctionalité dans le cas où on souhaite faire une union et où on a donc besoin de savoir si l'enveloppe convexe lui est égale.

Antoine Miné suggère l'ajout du *weak_ update* à la liste proposée par Bertrand Jeannet. Le CRI y est favorable. Cette opérateur permet de faire l'union entre les valeurs déjà associées à une dimension et un nouvel ensemble de valeurs. Il est utilisé pour modéliser la mise à jour d'un élément de tableau quand une unique location abstraite est définie pour tous les éléments du tableau.

La mise à jour parallèle de plusieurs dimensions, *parallel_ update*, a aussi été envisagée, mais non retenue pour le moment.

Les fonctions de conversion inter treillis sont envisagées, bien que l'utilisation de noms génériques pour les opérateurs en C ne permettent pas d'utiliser simultanement des treillis différents. Le problème est résolu dans PIPS en utilisant les noms génériques pour des pointeurs vers des fonctions et en initialisant ces pointeurs en fonction du contexte, mais il n'est pas sûr qu'il soit possible de d'empiler les structures qui contiennent ces pointeurs dans PIPS.

Il faut aussi prendre en compte les fonctions de conversion entre formats différents à treillis équivalents, par exemple, le passage d'une représentation dense à une représentation creuse pour un polyèdre.

Les opérateurs *new* et *free* seront supportés.

Les opérateurs de permutation de dimensions seront supportés.

La sérialisation et la désérialisation sont ASCII dans PIPS, qui gère le sharing via NewGen et qui évite le sharing dans l'analyseur sémantique, et binaire dans Astrée qui profite du sharing et qui stocke globalement toutes les informations sémantiques glanées pour un programme complet.

Les fonctions de lecture et d'impression sont nécessaires pour le debugging et pour la validation et pour le benchmarking. Le CRI souhaite que des noms de variables puissent être associés aux dimensions.

Deux niveaux d'interface, 1 et 2, ont été mentionnés sans être définis. La réunion et donc ce document concernet le niveau 1.

4 Problèmes

4.1 Problèmes divers

1. Sérialisation: binaire, ASCII ou les deux?
2. Existe-il in domaine abstrait *congruence* avec une implémentation en C?

   Philippe Granger et François Masdupuy ont-ils laissé quelque chose d'utilisable? Les travaux de François Masdupuy n'ont jamais pu être intégré dans PIPS

   Où en est-on avec les Z-polyèdres dans la Polylib?
3. Comment doit-on traiter les ensembles de points entiers contenus dans un polyèdre rationnel? Est-ce que cela fait partie de l'interface? Est-ce que cela défini un autre treillis, comme pour l'intersection avec pZ? Pour le moment, on intègre cette subtilité de manière discrète dans l'interface.

   Rappel: la projection, le test à vide, la minimisation,... sont différents
4. Support de la sous-approximation: au CRI d'en montrer l'intérêt lors d'une prochaine réunion. Pour le moment, on propose sur-approximation et exactitude.
5. Mécanisme de détection des *timeouts*: *threads concurrentes*? Exception? Discussion au sein de la Polylib pour avantages et inconvénients.
6. Typage des dimensions: *int, float, bool, char **. Plutôt typage global? Comment?
7. Utilisation de préfixes pour distinguer les différentes bibliothèques en C? Par exemple, OCT pour octagone, PPL pour Parma, HQ proposé par Duong.
8. Certaines heuristiques peuvent nécessiter des paramètres. Comment les passe-t-on?
9. Pour décider de la compatibilité entre deux arguments, comment peut-on se passer du mapping entre dimensions et adresses abstraites? En mettant ce test au niveau 2?
10. Souhaite-t-on avoir comme observateur un accès aux contraintes internes à l'objet? Compatibilité avec la généricité? Observateur renvoyant le plus petit objet contraignant une variable/dimension donnée?
11. Le CRI souhaite que tout type inclut une constante *xxx_undefined*. La discussion n'a pas été terminée sur ce point, l'étude de *is_bottom* ne s'y prêtant pas.

4.2 Types annexes

Au-delà du type de base du treillis lui-même, l'interface devraient prendre en compte:

1. le type *dimension*,
2. le type *expression*,
3. le type *expression affine*,
4. le type *prédicat affine*, e.g. plus grand, égal,...
5. le type *intervalle*,
6. le type *expression affine à coefficient constant intervalle*,
7. le type *expression affine à coefficients intervalles*, utiles pour traiter les flottants,
8. ...

La prise en compte des expressions générales est gênante pour l'utilisabilité puisque chaque analyseur définit différemment le type expression. Mais les expressions affines sont trop restrictives et reportent la linéarisation des expression au niveau supérieur alors qu'elle dépend du treillis utilisé.

4.3 Généricité des extracteurs

Il a semblé que les extracteurs dépendaient du treillis sous-jacent. Par exemple, on peut demander pour une variable ou pour un expression entière:

1. son signe,
2. sa parité,
3. sa valeur,
4. un intervalle de valeur,
5. l'intersection d'un intervalle de valeur et d'une congruence,
6. une expression affine égale,
7. une expression quelconque égale,
8. etc...

On peut aussi demander la valeur d'un prédicat, au moins d'un prédicat affine.

Nicolas Halbwachs suspecte que nous travaillons trop avec quelque chose qui est pratiquement une chaîne de treillis. Il serait bon, pour éviter les surprises, d'envisager les congruences et les BDDs pour les booléens.

4.4 Généricité des constructeurs et de l'interface

Les constructeurs proposés sont-ils trop orientés polyèdres? Les expressions affines sont-elles trop prises en compte par rapport aux expressions générales?

L'attention portée aux opérateurs de complexités exponentielles est-elle trop grande? Les conséquences en terme de lourdeur sont-elles acceptables pour des treillis simples comme les intervalles entiers?

5 L'opérateur *is_ bottom*

L'opérateur *is_ bottom*, qui renvoie, par exemple, vrai si l'ensemble des valeurs abstraites est vide, a été choisi du fait de sa simplicité. Simplicité apparente... Sa complexité est exponentielle et sa sémantique est différente en entier et en rationnel.

Il a permis de progresser sur la notion de *manager*. La réunion ne devait pas aboutir à une définition, mais à un ensemble de définitions possibles

5.1 Valeur retournée

1. Bertrand Jeannet propose de retourner un booléen ternaire, avec les valeurs {true, false, dontknow}. François Irigoin propose d'étendre cette valeur pour prendre en compte les débordements de capacité numérique et autres problèmes opérationnels.

2. au CRI, seulement des booléens sont retournés. La valeur *true* a pour sémantique *vrai*, la valeur *false* a pour sémantique je ne sais pas. Du coup, il faut définir une autre fonction, *is_ not_ bottom*.

3. Bertrand Jeannet et Nicolas Halbwachs trouve que cela rend l'interface difficile à comprendre puisque *not(is_ bottom)* est différent de *is_ not_ bottom*. Nicolas suggère d'appeler la fonction *is_ known_ bottom* pour lever l'ambiguité et d'ajouter *is_ known_ not_ bottom*.

4. Comme il faut aussi être *threadsafe*, passer des arguments opérationnels et récupérer des informations opérationnelles, il est proposé d'utiliser une structure de données opaque, appelée *manager*.

5. Le CRI aurait préféré pour des raisons de lisibilité que les prédicats aient des noms se terminant par *_p* comme *is_ known_ bottom_ p* parce que *is_ xxx* est utilisé en interne pour gérer les unions.

5.2 Les informations contenues dans le *manager* pour *is_ bottom*

Le contenu exact de *manager* doit être partiellement ignoré par l'utilisateur. Il dépend de la bibliothèque de *threads* utilisée et des besoins particuliers des implantations, comme celle de Bertrand Jeannet.

```
struct manager {
    struct internal * internal; // Bertrand Jeannet
    struct pool * pool;  // thread library
```

```
    int timeout;         // unity? millisecond?
    int max_object_size; // in abstract object size unit

    int algorithm;       // Algorithm selection: 0 is default algorithm
                         //                      MAX_INT is most accurate
                         //                        available
                         //                      MIN_INT is fastest
                         //                        available
                         //                      otherwise, algorithm n,
                         //                        with no speed or
                         //                        accuracy meaning for n

    bool flag_exact_wanted; // Return information about exactitude if
                            // possible
    bool flag_int_wanted;// Abstract set is a set of integer points, if
                         // not rational points. Meaning for intervals on
                         // floats?


    bool flag_time_out;  // timeout detected
    bool flag_out_of_space; // the size of the key (?) object exceeds
                         //    max_object_size
    bool flag_overflow;  // Magnitude overflow
    bool exception;      // If true, some operational exception was raised

    bool flag_exact;     // Result is mathematically exact or not:
                         // projection, satisfiability, union
}
```

Les opérateurs sur le *manager* n'ont pas été définis.

Duong Nguyen se demande si on ne devrait pas aussi signaler la non-implémentation d'une fonction par ce biais.

Bertrand Jeannet propose qu'une fonction appelée avec une exception levée dans l'argument manager fasse un retour immédiat. Son idée est d'éviter d'avoir à tester systématiquement le code de retour. François Irigoin préfèrerait qu'une erreur interne soit levée dans un tel cas, afin de faciliter la mise au point.

5.3 Autres opérateurs

Il reste à en discuter à la lumière des premières idées émises pour *is_bottom*. Comme un objet sera usuellement retourné, il sera difficile de généraliser l'idée du booléen ternaire *tbool*.

6 Conclusion

Différents niveaux d'interfaces sont prévus. Cette première réunion a été essentiellement consacrée au niveau le plus bas, le niveau 1.

Aucun programme de travail commun n'a été défini, si ce n'est de fournir un compte-rendu de réunion et de réfléchir à son contenu. Le CRI doit nettoyer les

bibliothèques et interfaces de PIPS, et faire un effort de clarification à propos des besoins de Sebastian Pop. Les problèmes de compatibilité avec l'existant, problèmes qui ont déjà été étudiés par Duong Nguyen, devront être pris en compte dans une prochaine réunion.

Le problème posé par la prise en compte des *threads* doit être exploré.

La prochaine réunion aura lieu début février, à Paris ou à Grenoble.

Annexe: Liste des opérateurs proposée par Bertrand Jeannet

Cette liste de déclarations ML n'a pas été (complètement) mise à jour en fonction des besoins exprimés par les uns et les autres. Les opérateurs n'ont pas non plus été discuté en détail, faute de temps, mise à part *is_ bottom*.

L'opérateur *minimize* réduit la taille de l'objet, tandis que l'opérateur *canonicalize* renvoie une représentation *canonique* qui peut être plus grande. L'existence d'une forme canonique n'est pas évidente pour tous les treillis.

L'opérateur *minimize*, appliqué avec l'option *int*, peut renvoyer une polyèdre rationnel contenant plus de points rationnels que son argument (expérience CRI, François Irigoin).

L'utilisation des tableaux comme arguments n'enthousiasme pas le groupe.

François Irigoin aurait voulu qu'on puisse demander la négation d'une condition linéaire pour ne pas avoir à traiter la diséquation. Bertrand Jeannet fait remarquer qu'il existe déjà suffisament d'opérateur pour l'exprimer et propose que ce soit remonté à l'interface de niveau 2. François Irigoin n'est pas convaincu que ce soit indépendant du treillis sous-jacent.

Il ne semble pas y avoir de support pour l'accélération, pour le *forget*, pour l'*undefined*.

L'exportation du type *generator* n'a pas semblée très générique. De plus *Point* pourrait être appelé *Vertex*.

Certains des commentaires faits durant l'exposé de Bertrand ont été perdus, mais beaucoup sont listés dans les sections précédentes (*weak_ update, parallel_ update*, forme affine avec coefficients de type intervalle,...).

FI: JE COMPTE SUR LES AUTRES PARTICIPANTS POUR BOUCHER LES TROUS. MERCI!

```
type num

module Linear = struct
  type linexpr = num array
  type affexpr = {
    lin : linexpr;
    cst : num;
  }
  type typcond = Sup | SupEg
  type cond = typcond * affexpr

  type generator
    | Point of affexpr
    | Ray of linexpr
    | Line of linexpr
end

type bound = num option
type interval = {
  inf : bound
  sup : bound
}
```

```
module type A = sig
  type t
    (** The type of abstract values *)

  (** *)
  val minimize : t -> unit
  val canonicalize : t -> unit

  (** Constructors and basic operations *)
  val bottom : int -> t
  val top : int -> t
  val meet : t -> t -> t
  val join : t -> t -> t
  val meet_linearcond : t -> Linear.cond -> t
  val meet_nonlinearcond : t -> NonLinear.cond -> t

  (** Tests *)
  val is_bottom : t -> bool
  val is_top : t -> bool
  val is_leq : t -> t -> bool
  val is_eq : t -> t -> bool
  val is_leq_linearcond : t -> Linear.cond -> bool

  (** Accessors *)
  val dim : t -> int
  val to_constraints : t -> Linear.cond list
  val to_generators : t -> Linear.generator list
  val to_bounds : t -> interval array
  val bound_linear_expr : t -> Linear.affexpr -> interval

  (** Widening *)
  val widening : t -> t -> t
    (** Standard widening *)
  val limited_widening : t -> t -> Linear.cond list -> t
    (** Widening with thresholds *)

  (** Projections, Embeddings, ... *)
  (* At the end *)
  val add_dims_and_embed : t -> int -> t
  val add_dims_and_project : t -> int -> t
  val del_dims : t -> int -> t
  (* Anywhere *)
  val add_permute_dims_and_embed : t -> int -> int array -> t
  val add_permute_dims_and_project : t -> int -> int array -> t
  val permute_del_dims : t -> int -> int array -> t

  (** Transformations *)
  val assign_var_linear : t -> int -> Linear.affexpr -> t
```

```
  val substitute_var_linear : t -> int -> Linear.affexpr -> t
  val assign_vars_linear : t -> (int * Linear.affexpr) array -> t
  val substitute_vars_linear : t -> (int * Linear.affexpr) array -> t

end
```