Action Concertée Incitative

SÉCURITÉ & INFORMATIQUE

APRON: Analyse de PROgrammes Numériques

Compte-rendu de la réunion du 1er février 2005: Interface commune

Introduction

Ce document réorganise et synthétise les compte-rendus des réunions du 16 décembre 2004, 1er février 2005 et 15 mars 2005.

Table des matières

Chapitre 1

Choix généraux

1.1 Sémantique d'une valeur abstraite

1. **Concrétisation d'une valeur abstraite.** Une valeur abstraite fournie par l'interface a pour concrétisation un sous-ensemble $X \subseteq \mathbb{N}^p \times \mathbb{R}^q$. Les variables sont donc typées, soit entières soit réelles.

2. . Les problèmes de précision et donc d'arithmétique modulaire pour les entiers 8, 16, 32 et 64 bits sont laissés à un niveau strictement supérieur à 1, voir dans l'analyseur lui-même au niveau de la sémantique concrète.

1.2 Niveaux d'interface

3. **Motivations.** 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.

Il s'agit d'assurer à la fois la performance des implantations et le confort de l'utilisateur, tout en évitant la duplication de code entre les différents librairies.

On propose donc de ne traiter au niveau 0 que les problèmes de performance (avantage algorithmique fort) et de reporter au niveau 1 les problèmes de confort, sachant que les problèmes de performance ne sont pas génériques, mais dépendent des domaines abstraits.

4. **Principe.** Il a été décidé d'établir différents niveaux d'interface.
   - Le niveau 0 est en prise directe avec la librairie sous-jacente (octogones, polyèdres) et suit les principes suivants :
     - On y place toutes les fonctions dont l'implémentation est spécifique au domaine abstrait et qui ne peuvent donc pas être partagées entre les librairies et .
     - L'interface est minimale : on en exclut les fonctions qui peuvent s'implémenter.à partir d'autres fonctions de niveau 0, **à moins qu'il n'y ait un avantage algorithmique fort** pour le laisser au niveau 0.
   - On réserve aux niveaux supérieurs les fonctions factorisables pour tous les domaines abstraits envisagés. Deux exemples envisagés :

1. L'appel automatique des opérations de redimensionnement et de permutations nécessaires pour calculer l'intersection $P1(x, y)$ avec $P2(y, z)$: ceci ne dépend pas du domaine abstrait considéré ; au pire, dans le cas de représentation interne creuses, il n'y a rien à faire.
2. L'abstraction d'expressions non-linéaires par des expressions linéaires d'intervalle.

On peut utiliser deux bibliothèques de niveau 0 pour les combiner de manière heuristique et obtenir une nouvelle bibliothèque de niveau 0.

Ci-dessous, points s'y rapportant :

Peut-on mettre au niveau 0 des fonctions spécifiques au domaine abstrait, par exemple une pré-analyse ?

Les produits cartésions pourraient être traités au niveau 1.

1.3 Langage de programmation

5. **Interface C** On définira au moins une version C de l'interface, qui serira de référence.

Le langage C a été choisi comme s'interfaçant aisément avec la plupart des langages (C++, Java, OCaml, Prolog, ...). C++ est moins adapté de ce point de vue. Par ailleurs, la plupart des librairies existantes sont en C (NewPolka, C3 (!?), octagones).

6. **Interface OCaml** L'IRISA et l'ENS sont intéressé par la définition d'une version OCaml.

Dans la suite, on ne discute que de l'interface C.

1.4 Formes normales des valeurs abstraites

La notion de forme(s) normale(s) a fait l'objet de vives discussions. Il s'agit de contrôler (abstraitement) la représentation interne des valeurs abstraites. Les besoins suivants ont été dégagés :

7. **Représentation minimale en terme d'occupation mémoire.** Cette notion existe dans les octogones (contraintes redondantes supprimées). Dans NewPolka cette notion n'est pas réellement fixée ; quelques possibilités :

1. Ensemble de contraintes non redondantes ;
2. Ensemble de contraintes *ou* de générateurs non-redondants, selon laquelle occupe le moins d'espace en mémoire ; c'est la solution correspondant à la définition ;
3. double représentation + matrice de saturation : c'est la seule notion de forme "normale" actuellement implantée dans NewPolka.

Dans les deux cas (octogones ou NewPolka), la forme minimale peut ne pas être adaptée aux calculs. Ainsi pour les octogones, la première chose à faire avant la plupart des opérations est de calculer la clôture de la forme minimale. Il en serait de même (dans une moindre mesure) pour la solution 2 dans NewPolka.

8. **Représentation canonique.** Il s'agit ici d'avoir une forme normale telle que deux valeurs sémantiquement égales ont exactement la même représentation.

La forme minimale précédente ne fournit pas forcément une représentation canonique. Par exemple, pour les polyèdres, il faut en outre normaliser la représentation de l'espace des égalités (ou des droites) et trier les contraintes (ou générateurs).

Problèmes se posent :
- Que faire lorsque les entiers ne sont que partiellement pris en compte ?
- Plus généralement, que faire lorsqu'on ne sait pas (ou on ne veut pas, pour des raisons de coût) la calculer ?

9. **Notion de réduction/approximation.** Dans certains cas, on veut contrôler assez finement la représentation interne, car elle a un impact sur la précision. Il s'agirait ici soit de simplifier la représentation, au prix éventuel d'une perte d'information (ex : on enlève des contraintes trop tarabiscotées, ou avec des coefficients de magnitude trop importante), soit d'affiner la représentation (ex de François pour la prise en compte des entiers, réduction dans le cas d'un domaine abstrait produit réduit de deux domaines abstraits) pour améliorer la précision d'opérations futures.

Une telle fonction sera fournie, paramétrée par un numéro d'algorithme. En outre, chaque opération sera paramétrée par la réduction/approximation à effectuer en entrée et en sortie de l'opération. Exemple(s) :
- Dans NewxPolka existent des versions strictes et paresseuses des opérations. La version stricte travaille toujours sur des (doubles) représentations minimisées, tandis que les versions paresseuses ne recourt à Chernikova que si indispensable. Le paramétrage de chaque opération permettra de maintenir ce type de réglale, sans polluer l'API par d'innombrables versions de la même opération sémantique.

1.5 Fonctionnalités et architecture générale de l'interface

10. **Compatibilité avec les threads.** L'interface permettra d'écrire des implantations compatibles avec les threads.

11. **Gestionnaire.** Un contexte d'appel, objet de type `manager_t*`, sera explicitement passé à chaque fonction afin d'assurer les point suivants.

La transmission de données globales spécifiques à chaque librairie (mémoire de travail, options non fournies via l'interface commune, ...), ce qui assure en particulier la compatibilité avec les threads.

La transmission des options (leviers de réglages des algorithmes,etc ...). En effet, 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.

la gestion des exceptions (récupération des exceptions) ; les débordements en capacité (entiers), en temps et en espace sont détectés, ou tout au moins, les mécanismes nécessaires à la récupération des incidents sont définis. Le retour d'un résultat mathématiquement exact peut être identifié à la demande de l'utilisateur (projection entière, union vs enveloppe convexe, test de satisfiabilité en entier,...).

Le contexte n'est pas intégré aux objets du domaine abstrait pour éviter d'avoir à faire une vérification de compatibilité des contextes.

C n'offrant pas de mécanisme d'exceptions (sauf via le compliqué `setjmp`), et celui-ci étant spécifiques à chaque langage (C++, OCaml, Java), les informations d'exception sont retournées uniquement via la structure `manager_t`.

Le contenu du contexte d'appel n'est pas visible directement : c'est un objet opaque, fermé, avec des méthodes. Des primitives de construction et d'observation seront définies et fournies. [bj] Ce choix présente aussi l'avantage de faciliter l'interfaçage avec un langage comme OCaml.

12. **Implantations partielles.** Les implantations partielles sont acceptées, mais elles doivent offrir toutes les signatures prévues pour permettre l'édition de lien et l'échec éventuellement en cas d'appel à une fonction non implantée.

Positionnement d'un flag `not_implemented` lorsqu'une fonction n'est pas implantée, et retour d'une valeur non spécifiée, le cas échéant (pointeur nul pour l'interface fonctionnelle ?)

13. **Gestion mémoire.** On n'implante pas de mécanisme automatique de ramasse-miettes (compteur de référence ou autre) pour l'interface C.

L'interface OCaml en revanche utilisera les mécanismes du runtime OCaml (idem pour Prolog ou autre).

14. **Signatures fonctionnelles et impératives.** Les signatures fonctionnelles et impératives sont toutes les deux supportées.

Questions en suspense :

1. impératif implique-t-il destructif ? [bj] oui
2. Des noms différents seront-ils utilisés ? Dans les octogones, utilisation d'un flag.
3. en mode impératif, les fonctions C renvoient-elles `void` (IRISA/VERIMAG) ou bien l'objet sur lequel un effet de bord a eu lieu (CRI/octogones d'Antoine Miné) ?
4. François Irigoin : suivre les conventions du package String de C et/ou les conventions du package *Set* de Pierre Jouvelot. Conventions à préciser.

Dans la proposition actuelle, on a choisi respectivement (oui,oui,non,non).

15. **Opérations n-aires.** Les opérations n-aires (ex : borne supérieure) sont supportées en utilisant des tableaux d'arguments.

La possibilité de passer plus de deux arguments n'est pas un simple confort. Elle 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. Idem pour les suites de projections. Enfin, elle a un impact sur la performance.

Le système `varargs` ne sera supporté que si un des participants en a le beosin (aucun pour l'instant). [bj] La version `vararg` est implantable en utilisant la version tableau, donc cela relève plutôt du niveau 1.

La définition d'une structure de liste propre à cette interface est exclue.

Le mécanisme de `varargs` est plus léger mais moins souple que le mécanisme de tableau parce qu'il impose un nombre d'arguments connu à l'avance. On décide donc d'utiliser des tableaux, et de ne proposer des *varargs* que si quelqu'un peut en montrer l'utilité. [bj] En outre, la version `vararg` étant implantable en utilisant la version tableau, elle relève plutôt du niveau 1.

16. **Types des objets et mode de passage des paramètres.** Quelques rappels sur les idiomes C :

1. choix entre passage par valeur et par référence (pointeur) :

   ⟨*bidon*⟩≡
   ```
   void toto(object_t o);  /* appel par valeur sur le type object_t */
   void toto(object_t* o); /* appel par référence sur le type object_t */
   ```

   Avantage de l'appel par référence : plus rapide si `sizeof(object_t)` est sensiblement plus grand que la taille d'un scalaire (4 ou 8 octets). Inconvénient : casse-bonbon pour le programmeur de devoir écrire `toto(&x)` au lieu de `toto(x)`.

2. choix similaire pour la valeur retournée :

   ⟨*bidon*⟩+≡
   ```
   object_t toto();  /* retour par valeur */
   object_t* toto(); /* valeur de retour allouée par toto et à libérer par
                        l'appelant si nécessaire */
   ```

   Là, le retour par référence nécessite une allocation, donc plutôt moins bien point de vue efficacité !

3. type abstrait en C = type pointeur.

La dernière remarque justifie la signature suivante dans l'interface (*c.f.* §5.3), *si l'on choisit un passage par valeur* :

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet(manager_t* man, abstract_t* a1, abstract_t* a2);
```

En effet tous les types impliqués sont considérés comme abstraits. Ici donc, on a un passage par valeur sur des types pointeurs, *et non un passage par référence sur les types* `manager_t` *et* `abstract_t`. C'est d'une certaine manière une question d'interprétation, bien sûr, mais en pratique si tous les objets sont déclarés, construits, manipulés avec un type `object_t*`, il n'y a jamais de déréférencement à faire et le programmeur n'a aucune question à se poser (c'est ce qui se passe dans la librairie de BDDs CUDD).

Maintenant, si le type `lincons_t` des contraintes linéaires utilisé dans `abstract0.h` est abstrait (généré à partir d'un type utilisateur, *c.f.* discussion dans §4.2), les signatures suivantes sont "imposées" (si passage par valeur) :

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet_lincons(manager_t* man,
                                  abstract_t* a, lincons_t* c);
abstract_t* abstract_meet_lincons_array(manager_t* man,
                                        abstract_t* a,
                                        lincons_t** tlincons, int size);
```

Une contrainte linéaire étant de type `lincons_t*`, les tableaux de contraintes linéaires sont de type `lincons_t**`.

Mais si le type `lincons_t` utilisés dans ces fonctions est utilisateur (public), on pourrait *aussi* utiliser (toujours si passage par valeur)

⟨*bidon*⟩+≡
```
abstract_t* abstract_meet_lincons(manager_t* man,
                                  abstract_t* a, lincons_t c);
abstract_t* abstract_meet_lincons_array(manager_t* man,
                                        abstract_t* a,
                                        lincons_t* tlincons, int size);
```

Dans l'interface proposée, les conventions suivies sont :
- Appel et retour par valeur, sauf pour les paramètres résultats (taille de tableaux retournés) ;
- Les types des objets (`abstract_t*, linexpr_t*, lincons_t*, ray_t*`) sont en général des types pointeurs, sauf pour :
  - Les nombres (`coeff_t`) ;
  - Les intervalles (`interval_t, boundedinterval_t`)

  Ces deux derniers types sont considérés comme des types utilisateurs *c.f.* §3, et leur taille (en terme de `sizeof()`) reste petite (mais ces structures peuvent bien sûr contenir des pointeurs vers des zones de taille importante, *e.g.*. nombres multi-précision).

17. **Retour d'arguments multiples.** Quelle politique lorsque plusieurs arguments doivent être retournés ? Par exemple, si une fonction retourne un tableau d'objets de taille non connue à l'avance, il faut retourner un pointeur sur un tableau plus la taille du tableau.

Essaye-t-on d'unifier l'interface ? En C, lorsque `toto(x)` doit retourner deux arguments `a` et `b`, on a 3 solutions ;

⟨*bidon*⟩+≡
```
toto(x,&a,&b); /* paramètres résultats, pasq très joli; */

a=toto(x,&b);  /* une valeur retournée, l'autre en paramètre résultat;
                  guère plus joli, et très moche si a et b ont une
                  signification "symmétriques"
                */

struct titi { a_t a; b_t b; };
titi=toto(x); /* creation d'un nouveau type, parfois
                 juste pour une seule fonction */
```

Ah, ces `*%$č&#` de C, C++, Java !

Pour l'interface OCaml, qu'on essayera de générer un peu automatiquement à partir de l'interface C, à noter que CamlIDL sait assez bien traiter les paramètres résultats, en les transformant en valeurs retournées dans un n-uplet.

18. **Représentation interne des nombres.** Le type numérique utilisé en interne dans une librairie sous-jacente est défini à la compilation, e.g. via une option `-D`, et/ou à l'édition de lien (comme dans PIPS, type *Value, et dans Polka, type* pkint).

Un type numérique utilisateur est fourni par l'interface, *c.f.* §3.

19. **Nommage des fonctions.** On utilisera des préfixes pour distinguer les différentes bibliothèques en C ? Par exemple, OCT pour octagone, PPL pour Parma, HQ proposé par Duong. Préfixes envisagés : POLKA, C3, PIPS, POLYLIB..

Ne pas utiliser de macros mais une moulinette de désambiguation ? ! ? [bj] Comprends pas trop "moulinette'.

20. **Sérialisation/désérialisation.** La sérialisation/désérialisation des objets (valeurs abstraites, contraintes, ...) sera fournie par l'interface.

Si la sérialisation se fait sur les types utilisateurs (contraintes notamment), alors possibilité d'échanger entre plusieurs librairies. Si elle se fait sur un type abstrait, ce n'est plus le cas (sans compter le problème de la représentation des coefficients dans les types internes, dépendant d'options de compilation de la même librairie).

21. **Conversions et super-treillis.** Afin de faciliter les conversions entre valeurs abstraites de domaines différents, on peut considérer un super-treillis et n'implanter que les conversions d'un domaine abstrait vers ce super-treillis.

En l'absence d'informations de congruence, les polyèdres convexes suffisent pour tous les treillis abstraits envisagés (intervalles, égalités, octogones, templtate constraints, octaèdres, polyèdres convexes). Toutefois, si on veut ajouter les égalités polynomiales de degré borné ([Seidl]), ce n'est plus vrai.

Si on rajoute les congruences, il faut plus. Presburger est un peu candidat, mais quid des réels ?

Chapitre 2

Contexte d'appel et fichier `manager.h`

Ce chapitre définit le fichier `manager.h` se rapportant à la définition du contexte d'appel.

⟨*manager.h*⟩≡
```
/* ********************************************************************** */
/* manager.h : global manager passed to all functions */
/* ********************************************************************** */

/* ********************************************************************** */
/* I. Types */
/* ********************************************************************** */
```

Le type `manager_t` est structuré comme suit :

⟨*manager.h*⟩+≡
```
/* Manager (opaque type) */
typedef struct manager_t {
  char* library;                  /* name of the effective library */
  struct internal_t* internal;    /* library dependent,
                                     should be different for each thread
                                     (working space) */
  struct option_t* option;        /* Options (in) */
  struct result_t* result;        /* Exceptions and other indications (out) */
} manager_t;
```

2.1 Options

On synthétise les décisions prises (ou en cours) avant de refléter les discussions à ce sujet.

22. **Options associées à chaque opération.** À chaque opération (non triviale) sera associée un objet du type suivant :

⟨*manager.h*⟩+≡

```
/* Option associated to each function (public type) */
typedef struct foption_t {
  int algorithm;
  /* Algorithm selection :
     - 0 is default algorithm;
     - MAX_INT is most accurate available;
     - MIN_INT is most efficient available;
     - otherwise, no accuracy or speed meaning
  */
  int approx_before;
  int approx_after;
  /* Related to the notion of approximation/reduction.
     Indicates which kind of "approximation" may be performed on the
     argument(s) before the algorithm, and on the result delivered
     by the algorithm. 0 is default behaviour. */
  int timeout; /* unit!? */
  /* Above the given computation time, the function may abort with the
     exception flag flag_time_out on.
  */
  int max_object_size; /* in abstract object size unit. */
  /* If during the computation, the size of some object reach this limit, the
     function may abort with the exception flag flag_out_of_space on.
  */
} option_t
```

On a choisi de spécifier ce type d'options fonction par fonction, afin d'éviter à avoir à repositionner des options globales entre chaque appel.

23. **Taille abstraite.** Une notion de taille abstraite d'objet est définie. Disposer d'une taille concrète (en octets) est envisageable mais potentiellement très coûteux à évaluer (notamment lors de l'emploi de nombres en multi-précision). or il est souhaitable que le mécanisme d'"out_of_space" ne pénalise pas trop les performances.

24. **Mécanisme de détection des *timeout*** : threads concurrentes ? Exception ? Discussion au sein de la Polylib pour avantages et inconvénients. [bj] Apparemment, on a opté pour exception.

25. **Options paramétrées.** Certaines heuristiques peuvent nécessiter des paramètres. Comment les passe-t-on ? En ajoutant des champs dans *manager* ou dans le champs *internal* de *manager* ? [bj] Je suis en faveur de cette solution : les paramètres des heuristiques dépendent fortement du domaine abstrait

26. **Ensemble des options.**

⟨*manager.h*⟩+≡
```
/* Options (in) (opaque type) */
typedef struct option_t {
  foption_t minimize;
  foption_t canonicalize;
  foption_t approximate;
  foption_t meet;
  foption_t join;
  foption_t meet_lincons;
  foption_t meet_intlincons;
  foption_t assign_linexpr;
  foption_t substitute_linexpr;
  foption_t assign_intlinexpr;
  foption_t substitute_intlinexpr;
  foption_t widening;
  bool_t flag_exact_wanted; /* return information about
                               exactitude if possible */
  bool_t flag_best_wanted;  /* return information about
                               best correct approximation if possible */
} option_t;
```

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

Idem pour un résultat représentant la meilleure approximation correcte de l'opération dans le domaine abstrait considéré.

27. **Question.** [bj] Dans les deux types précédents, on a choisi une organisation mémoire à la C, dans laquelle une structure hiérarchique est mise à plat. Ceci est est plus pratique en C, car ne nécessite ni constructeur ni manipulation de pointeurs, mais pour l'interfaçage avec OCaml ou Java, c'est nettement moins pratique.

2.2 Exceptions

28. **Type** le type `exception_t` est défini comme suit :

⟨*manager.h*⟩+≡
```
/* Exceptions (public type) */
typedef enum exception_t {
  EXC_NONE=0,           /* no exception detected */
  EXC_TIMEOUT=1,        /* timeout detected */
  EXC_OUT_OF_SPACE=2,   /* out of space detected */
  EXC_OVERFLOW=3        /* magnitude overflow detected */
  EXC_NOT_IMPLEMENTED=4 /* not implemented */
} exception_t;

/* Boolean lattice*/
typedef enum tbool_t {
  tbool_false=0,
  tbool_true=1,
  tbool_top=2,   /* don't know */
} tbool_t;

/* Exceptions and other indications (out) (opaque type) */
typedef struct result_t {
  exception_t exception; /* exception returned */
  tbool_t flag_exact; /* result is mathematically exact or not
                         or don't know */
  tbool_t flag_best;  /* result is best correct approximation or not
                         or don't know */
} result_t;
```

Remarque importante : dans la précédente proposition, on avait des flags d'exception (un par exception) et se posait une question non tranchée : suppose-t-on que les flags d'exceptions sont implicitement remis à faux au début de chaque appel de fonction, ou faut-il le faire manuellement ? Ici, l'utilisation d'un type énuméré résoud le choix en faveur de la première option.

29. **Discussion :** Le CRI souhaite que tout type utilisé comme résultat d'une fonction inclut une constante *xxx_undefined*. La discussion n'a pas été terminée lors de la première réunion sur ce point, l'étude de *is_bottom* ne s'y prêtant pas.

La valeur *undefined* n'apparait pas dans les domaines abstraits.

Discussion à reprendre :

- l'information ne devrait-elle pas être portée dans le *manager* ? Non pour le CRI qui se place dans une perspective de mise au point.
- *undefined* n'est-il pas *top* ? Le CRI doit effectivement utiliser parfois *undefined* comme *top* pour alléger l'implémentation. La sémantique d'*undefined* est-elle bien définie ? Top ? Not implemented ? Exception occured ? Utilisation des codes d'exception ?
- *fail_with* ? ! ?
- Jérôme : utilisation pour des effets de bord détectés mais non traités.
- propagation des erreurs ? des exceptions ? remises à zéro du *manger* ?

2.3 Fonctions d'accès

⟨*manager.h*⟩+≡

```
/* ********************************************************************** */
/* II. Functions */
/* ********************************************************************** */

/* Constructor and destructor for internal */
internal_t* manager_internal_alloc();
void manager_internal_free(internal_t* internal);

/* Constructor and destructor for manager */
manager_t* manager_alloc(internal_t* internal);
void manager_free(manager_t* man);
  /* also free internal field if it is not yet put to NULL */

/* Reading fields */
internal_t* manager_get_internal(manager_t* man);

option_t manager_get_minimize(manager_t* man);
option_t manager_get_canonicalize(manager_t* man);
option_t manager_get_meet(manager_t* man);
option_t manager_get_join(manager_t* man);
option_t manager_get_meet_lincons(manager_t* man);
option_t manager_get_meet_intlincons(manager_t* man);
option_t manager_get_assign_linexpr(manager_t* man);
option_t manager_get_substitute_linexpr(manager_t* man);
option_t manager_get_assign_intlinexpr(manager_t* man);
option_t manager_get_substitute_intlinexpr(manager_t* man);
option_t manager_get_widening(manager_t* man);

bool_t manager_get_exact_wanted(manager_t* man);
bool_t manager_get_best_wanted(manager_t* man);
```

```
exception_t manager_get_exception(manager_t* man);
tbool_t manager_get_flag_exact(manager_t* man);
tbool_t manager_get_flag_best(manager_t* man);

/* Settings fields */
void manager_set_internal(manager_t* man, internal_t* internal);

void manager_set_minimize(manager_t* man, foption_t foption);
void manager_set_canonicalize(manager_t* man, foption_t foption);
void manager_set_meet(manager_t* man, foption_t foption);
void manager_set_join(manager_t* man, foption_t foption);
void manager_set_meet_lincons(manager_t* man, foption_t foption);
void manager_set_meet_intlincons(manager_t* man, foption_t foption);
void manager_set_assign_linexpr(manager_t* man, foption_t foption);
void manager_set_substitute_linexpr(manager_t* man, foption_t foption);
void manager_set_assign_intlinexpr(manager_t* man, foption_t foption);
void manager_set_substitute_intlinexpr(manager_t* man, foption_t foption);
void manager_set_widening(manager_t* man, foption_t foption);

void manager_set_exact_wanted(manager_t* man, bool_t* b);
void manager_set_best_wanted(manager_t* man, bool_t* b);
```

Chapitre 3

Représentation des coefficients et fichier coeff.h

30. **Représentation des nombres dans les librairies sous-jacentes et types abstraits.** Elle est entièrement libre.

- Les entiers peuvent être représenté par des `int`, `long int`, `long long int`, ou des entiers multi-précisions (d'une librairie quelconque).
- les réels peuvent être représentés par des rationnels sur les entiers précédents, ou par des flottants (`float`, `double` ou flottantsq multi-précisions), ou même des intervalles de flottants (!? pas sûr, attends confirmation de l'ENS ; sans doute nécessaire pour les conversions).

31. **Coefficients et leurs représentations dans les types utilisateurs.** Il est important de distinguer le domaine concret des coefficients et leur représentation dans les expressions, contraintes, générateurs, ...

Les dimensions étant soit entières, soit réelles, on peut considérer qu'il en est de même pour les coefficients. En concret, les entiers étant un sous-ensemble des réels, on peut se limiter aux réels. Mais cette propriété d'inclusion n'est pas vérifiée au niveau de la représentation (voir exemple précédent). Typiquement, un entier multi-précision n'est pas toujours représentable de manière exacte par un `float`.

3.1 Représentation des nombres

32. **Proposition de BJ** Je propose comme base de discussion le type suivant :

⟨*coeff.h*⟩≡

```
/* ********************************************************************** */
/* I. Coefficients  */
/* ********************************************************************** */

typedef enum coeff_discr_t = {
  COEFF_RAT,   /* rationnel avec multiprécision de GMP */
  COEFF_DOUBLE /* flottant avec double */
}
typedef coeff_t {
  coeff_discr_t discr ;
```

```
  union {
    gmpq_t rat;
    double flt;
  };
} coeff_t;

/* follows all the necessary operations on numbers of type coeff_t,
   using GMP conventions */

coeff_add(coeff_t a, coeff_t b, coeff_t c);

/* ... */
```

Commentaires :

- Les rationnels multi-précisions incluent le cas des rationnels sur des entiers à précision fixe (`int`, `long int`, etc).
- `double` inclut `float`. Toutefois, pour plus de généralité, on pourrait même prendre des flottants multi-précisions. Antoine Miné autorise `mpfr` dans ses octogones, mais cette version est-elle réellement utilisée ?
- `coeff_add` doit-il vérifier la compatibilité de ses arguments in ? Doit-il automatiquement effectuer les conversions (au prix éventuel d'une perte d'information) ?
- On autorise ici que chaque coefficient apparaissant dans une expression puisse avoir un type différent (rationnel ou flottant). On pourrait être moins flexible en imposant dans chaque expression un type unique pour les coefficients (le champ de type `coeff_discr_t` serait alors associé à l'expression). Avantage : lors de l'addition de deux expressions "2x+3y+4z" et "x+y", pas besoin de vérifier à tout bout de champ la compatibilité des arguments.

[bj] Je pense qu'il s'agit d'un point important pour la prochaine réunion. Antoine a plus d'expérience que moi avec sa librairie d'octogones, et l'ENS en général pour l'utilisation des flottants/intervalles de flottants

3.2 Intervalles

33. **Intervalles non bornés** (pour les expressions linéaires d'intervalles, pour les extracteurs de propriétés)

⟨*coeff.h*⟩+≡

```
/* ********************************************************************** */
/* II. Intervals  */
/* ********************************************************************** */

typedef struct bound_t = {
  bool_t is_bound; /* false means no bound (infinity) */
  coeff_t bound;
} bound_t;

typedef struct interval_t = {
  bound_t inf;
  bound_t sup;
} boundinterval_t;
/* QUESTION : should we consider also open intervals? */
```

34. **Manipulation d'intervalles.** Fournit-on également les opérations d'arithmétique d'intervalle ?

⟨*coeff.h*⟩+≡

```
/* Operations on interval
void interval_add(interval_t a, interval_t b, interval_t c);
/* or */
void interval_add(interval_t* a, interval_t* b, interval_t* c);
/* ... */
```

Chapitre 4

Sémantique concrète et fichier expr0.h

⟨*expr0.h*⟩≡
```
/* ************************************************************************** */
/* expr0.h : datatypes for dimensions, expressions and constraints */
/* ************************************************************************** */

#include "coeff.h"
```

4.1 Dimensions

35. **Synthèse des décisions prises.**

1. Au niveau 0, la concrétisation d'une valeur abstraite est un sous-ensemble $X \subseteq \mathbb{N}^p \times \mathbb{R}^q$.
2. Les dimensions sont numérotées de 0 à $p+q-1$ et sont donc typées. Une dimension i est entière si $0 \leq i < p$, réelle sinon.
3. La dimension (p, q) de l'espace $\mathbb{N}^p \times \mathbb{R}^q$ dans lequel un sous-ensemble abstrait est plongé est accessible par un opérateur.
4. Des arguments sont compatibles au niveau 0 s'ils ont tous la même dimension.
5. Les opérations peuvent ignorer le caractère entier d'une dimension, s'il en résulte une surapproximation (exemple : ce que fait NewPolka).
6. La gestion de la liaison entre dimension et zone de mémoire abstraite (e.g. identificateurs dans les cas simples, adresses mémoires, ensemble d'adresses mémoire) est déléguée au niveau 1.

⟨*expr0.h*⟩+≡
```
/* datatype for dimensions */
typedef int dim_t
```

4.2 Expressions linéaires et extensions

36. **Duplication type utilisateur/type interne.** L'interface définit un type utilisateur d'expression, qui ne correspond en général pas au type interne utilisé par une librairie sous-jacente. Deux solutions sont possibles, que l'on illustre sur une fonction d'affectation du type `abstract_t*`
`abstract_assign_linexpr(abstract_t* a, dim_t dim, linexpr_t`
`expr)`.

1. La fonction `abstract_assign_linexpr` prend en entrée le type public des expressions `linexpr_t` et effectue la conversion vers le type interne à la volée ;
2. Il existe dans l'interface un type interne (opaque) explicite, avec les fonctions de conversions nécessaires (ou les constructeurs nécessaires), et `abstract_assign_linexpr` prend en entrée le type interne.

Le choix 1 est plus pratique d'un point de vue utilisateur, mais les conversions peuvent être coûteuses, d'autant qu'elles incluent également des conversions de représentations des coefficients. Le choix 2 est plus économique, mais alourdit l'interface et son utilisaton.

Questions liées :

1. On a considéré le cas du passage d'objet à une fonction interne, mais il y a aussi le cas du retour à considérer. [bj] A mon avis, autant que ce soit homogène : si on passe des types internes, autant retourner des types internes, même si souvent (mais pas toujours) la valeur retournée va être examinée par l'appelant, ce qui nécessite une conversion vers le type utilisateur.
2. Si on fournit la sérialisation des expressions, faut-il la fournir à la fois pour le type utilisateur et le type interne ?

37. **Type utilisateur pour les expressions linéaires.** Proposition d'une représentation dense :

⟨*expr0.h*⟩+≡
```
/* ********************************************************************** */
/* III. Linear and affine expressions  */
/* ********************************************************************** */

/* Datatype for linear expressions */
typedef struct linexpr_t {
  coeff_t cst;    /* constant coefficient */
  coeff_t* coeff; /* coefficients associated to dimensions 0..size-1 */
  int size;       /* number of dimensions */
} linexpr_t;
```

Mais la représentation creuse suivante est mieux adaptée si le nombre de dimension est importante, ou pour les octogones si on veut détecter les contraintes simples "octogonales". À noter, *pour le niveau 1, seul ce type de repésentation sera adapté.* En revanche, bien moins pratique à manipuler ! On peut aussi fournir un type union des deux.

⟨*bidon*⟩≡
```
/* Sparse datatype for linear expressions */
typedef struct linterm_t {
  coeff_t coeff;
  dim_t dim; /* if dim=-1, constant coefficient */
} linterm_t;
typedef struct slinexpr_t {
  linterm_t* linterms; /* invariant : sorted in increasing order wrt dimensions */
  int size;            /* number of terms */
} slinexpr_t;
```

Enfin, si on veut typer les dimensions dans les expressions, il faut remplacer `size` par `intdim` et `realdim`.

38. **Manipulation d'expressions.** Doit-on fournir des opérations de manipulation d'expressions ?

39. **Type utilisateur pour les expressions linéaires d'intervalle**

Ces expressions permettent d'abstraire des expressions non-linéaires, et aussi de prendre en compte la sémantique spécifique des flottants.

⟨*expr0.h*⟩+≡
```
/* Datatype for linear interval expressions */
typedef struct intlinexpr_t {
  interval_t cst;    /* constant coefficient */
  interval_t* coeff; /* coefficients associated to dimensions 0..size-1 */
  int size;                 /* number of dimensions */
}
```

4.3 Contraintes linéaires

40. **Type utilisateur pour les contraintes linéaires.**

⟨*expr0.h*⟩+≡

```
/* ********************************************************************** */
/* IV. Linear constraints */
/* ********************************************************************** */

/* Datatype for type of constraints */
typedef enum constyp_t = {
  eq,    /* equality constraint */
  supeq, /* >= constraint */
  sup    /* > constraint */
} constyp_t;

/* Represents the constraint "expr constyp 0" */
typedef struct lincons_t {
  constyp_t constyp; /* type of constraint */
  linexpr_t* expr;   /* expression */
} lincons_t;

/* Represents the constraint "intexpr constyp 0" */
typedef struct intlincons_t {
  constyp_t constyp; /* type of constraint */
  intlinexpr_t* expr;   /* expression */
} intlincons_t;
```

4.4 Générateurs

41. Obtenir les générateurs d'une valeur abstraite est utile pour certaines applications (génération de code). Par ailleurs, la connaissance des rayons permet de savoir quelles sont les directions infinies dans une valeur abstraite.

42. **Type utilisateur pour points et rayons.**

⟨*expr0.h*⟩+≡

```
/* ********************************************************************** */
/* V. Generators */
/* ********************************************************************** */

/* Datatype for a vertex */
typedef linexpr_t vertex_t;

/* Datatype for a ray */
typedef linexpr_t ray_t;
  /* constant coefficient is ignored in both types */
```

[bj] Ici, souci de ne pas dupliquer les types. Points et générateurs ne sont pas inclus dans un type "generator", ce qui veut dire qu'on ne peut pas les mélanger dans un même tableau..

4.5 Expressions et contraintes non linéaires

43. Le cas des contraintes linéaires n'a pas été encore abordé. Une solution viable semble être d'abstraite, au niveau 1 de l'interface, les expressions non-linéaires par des expressions linéaires d'intervalle. Il n'est pas sûr que l'on puisse faire beaucoup mieux en laissant chaque domaine abstrait se débrouiller avec les expressions non linéaires (*i.e.*, le code de conversion vers les expressions linéaires d'intervalle risque d'être dupliqué).

4.6 Congruences

44. On ne dispose d'aucune implémentation des congruences, et donc d'aucun exemple sous la main. Le type `ray_t` permet en princie de représenter des informations de congruence sur les entiers.

Chapitre 5

Interface du domaine abstrait : fichier `abstract0.h`

⟨*<abstract0.h*⟩≡
```
/* ********************************************************************** */
/* abstract.h : generic operations on generic numerical
/* ********************************************************************** */

#include "manager.h"
#include "expr0.h"
```

5.1 Gestion mémoire, Représentations, Entrées/Sorties

⟨*abstract0.h*⟩≡
```
/* ********************************************************************** */
/* I. General management */
/* ********************************************************************** */
```

45. **Initialisation.** Elle se fait en allouant un manager.

46. **Gestion mémoire**

⟨*abstract0.h*⟩+≡
```
/* ============================================================ */
/* I.1 Memory */
/* ============================================================ */

abstract_t* abstract_copy(manager_t* man, abstract_t* a);
  /* Return a copy of an abstract value, on
     which destructive update does not affect the initial value. */

void abstract_free(manager_t* man, abstract_t* a);
  /* Free all the memory used by the abstract value */

int abstract_size(manager_t* man, abstract_t* a);
  /* Return the abstract size of an abstract value (see manager_t) */
```

47. Control of internal representation

⟨*abstract0.h*⟩+≡

```
  /* ========================================================== */
  /* I.2 Control of internal representation */
  /* ========================================================== */

  void abstract_minimize(manager_t* man, abstract_t* a);
    /* Minimize the size of the representation of a.
       This may result in a later recomputation of internal information.
    */

  void abstract_canonicalize(manager_t* man, abstract_t* a);
    /* Put the abstract value in canonical form. (not yet clear definition) */

  void abstract_approximate(manager_t* man, abstract_t* a, int algorithm);
    /* Perform some transformation on the abstract value, guided by the
       field algorithm.

       The transformation may lose information.  The argument "algorithm"
       overrides the field algorithm of the structure of type foption_t
       associated to abstract_approximate (commodity feature). */

  tbool_t abstract_is_minimal(manager_t* man, abstract_t* a);
  tbool_t abstract_is_canonical(manager_t* man, abstract_t* a);
```

Les propriétés de la forme canonique ne sont pas encore bien claires (*c.f.* §1.4). On aimerait que par exemple la sérialisation en binaire d'un objet canonique soit aussi canonique.

48. Impression.

⟨*abstract0.h*⟩+≡

```
  /* ========================================================== */
  /* I.3 Printing */
  /* ========================================================== */

  void abstract_print(FILE* stream,
                      manager_t* man,
                      abstract_t* a,
                      char*(*name_of_dim)(dim_t dim));
    /* Print the abstract value in a pretty way, using function
       name_of_dim to name dimensions */

  void abstract_printdiff((FILE* stream,
                           manager_t* man,
                           abstract_t* a1, abstract_t* a2,
                           char*(*name_of_dim)(dim_t dim));
    /* Print the difference between a1 (old value) and a2 (new value),
       using function name_of_dim to name dimensions.
       The meaning of difference is library dependent. */

  void abstract_dump(FILE* stream, manager_t* man, abstract_t* a);
    /* Dump the internal representation of an abstract value,
       for debugging purposes */
```

49. **Précisions sur les fonctions d'impressions.** Le format des fonctions d'impression est propre à chaque librairie : aucune syntaxe n'est imposée, ne serait-ce que parce que la représentation interne des nombres n'est pas unifiée.

Si l'on désire une syntaxe uniforme, il faut d'abord convertir en contraintes/générateurs utilisateur.

50. **Sérialisation**

⟨*abstract0.h*⟩+≡
```
/* ========================================================== */
/* I.4 Serialization */
/* ========================================================== */

int abstract_serialize_raw(FILE* stream, manager_t* man, abstract_t* a);
  /* Output the abstract value in raw binary format to the stream and return
     the number of bytes written */

abstract_t* abstract_deserialize_raw(FILE* stream, manager_t* man, int* size);
  /* Return the abstract value read in raw binary format
     from the input stream and store in size the number of bytes read */
```

La sérialisation/désérialisation en binaire, comme déjà mentionné, ne fonctionne que pour la même librairie, compilée avec les mêmes options de représentation interne des nombres.

Antoine Miné, dans ses octogones, a des fonctions de sérialisation vers des zones mémoires (au lieu d'un fichier). C'est plus général, mais plus poison aussi, car il faut connaître la taille nécessaire à l'avance, ou alors réallouer en cours de route.

5.2 Constructeurs, accesseurs, tests et extraction de propriétés

51. **Basic constructors**

⟨*abstract0.h*⟩+≡
```
/* ********************************************************************** */
/* II. Constructor, accessors, tests and property extraction */
/* ********************************************************************** */

/* ========================================================== */
/* II.1 Basic constructors */
/* ========================================================== */

/* We assume that dimensions [0..intdim-1] correspond to integer variables, and
   dimensions [intdim..intdim+realdim-1] to real variables */

abstract_t* abstract_bottom(manager_t* man, int intdim, int realdim);
  /* Create a bottom (empty) value */

abstract_t* abstract_top(manager_t* man, int intdim, int realdim);
  /* Create a top (universe) value */
```

On pourrait aussi rajouter les deux fonctions suivantes (qui peuvent être vues comme des fonctions de conversion).

⟨*abstract0.h*⟩+≡
```
  abstract_t* abstract_of_box(manager_t* man,
                              int intdim, int realdim,
                              interval_t* tinterval);
    /* Abstract an hypercube defined by the array of intervals
       of size intdim+realdim */

  abstract_t* abstract_of_lincons_array(manager_t* man,
                                        int intdim, int realdim,
                                        lincons_t** tlincons, int size);
    /* Abstract a convex polyhedra defined by the array of linear constraints
       of size size */
```

`abstract_of_box` permet d'abstraire assez aisément un point de l'espace.

`abstract_of_lincons_array` est redondant si l'on dispose de `abstract_meet_lincons_array` (ou autre nom, `abstract_meet_polyhedra`), puisqu'il suffit alors de créer la valeur top puis d'appeler `abstract_meet_lincons_array` avec l'ensemble des contraintes.

52. Accessors

⟨*abstract0.h*⟩+≡
```
  /* ============================================================ */
  /* II.2 Accessors */
  /* ============================================================ */

  /* Dimensions */
  int abstract_dimension(manager_t* man, abstract_t* a,
                         int* intdim, int* realdim);
    /* Store the number of integer and real dimensions in intdim and realdim,
       and return their sum. */
```

Ancienne proposition, *c.f.* problème du retour de plusieurs objets.

⟨*bidon*⟩≡
```
  int abstract_integer_dimension(manager_t* man, abstract_t* a);
  int abstract_real_dimension(manager_t* man, abstract_t* a);
  int abstract_dimension(manager_t* man, abstract_t* a);
```

53. Tests

⟨*abstract0.h*⟩+≡

```
/* ============================================================ */
/* II.3 Tests */
/* ============================================================ */

/* If any of the following functions returns tbool_top, this means that
   an exception has occured, or that the exact computation was
   considered too expensive to be performed (according to the options).
   The flag exact and best should be cleared in such a case. */

tbool_t abstract_is_bottom(manager_t* man, abstract_t* a);
tbool_t abstract_is_top(manager_t* man, abstract_t* a);

tbool_t abstract_is_leq(manager_t* man, abstract_t* a1, abstract_t* a2);
  /* inclusion check */

tbool_t abstract_is_eq(manager_t* man, abstract_t* a1, abstract_t* a2);
  /* equality check */

tbool_t abstract_sat_lincons(manager_t* man, abstract_t* a, lincons_t* a);
  /* does the abstract value satisfy the linear constraint? */

tbool_t abstract_sat_interval(manager_t* man, abstract_t* a,
                              dim_t dim, interval_t interval);
  /* is the dimension included in the interval in the abstract value? */
```

54. **Le problème du test du vide avec les entiers.** Que doit retourner `abstract_is_bottom` si les entiers ne sont que partiellement pris en compte ? Si un polyèdre n'est pas vide en rationnel, on peut retourner :

- `tbool_false` avec le flag `flag_exact` à `tbool_false` ou `tbool_top` ;
- `tbool_true` avec le flag `flag_exact` à `tbool_false` ou `tbool_top` ;
- `tbool_top` avec le flag `flag_exact` à n'importe quelle valeur
- ...

En fait, la redondance illustrée ci-dessus entre la valeur retournée et le flag `flag_exact` est générale à tous les tests.

Comment faire si on veut juste savoir si un polyèdre sur des dimensions déclarées entières est vide ou non au sens des rationnels ? Dans NewPolka, on utilise quand-même le typage des dimensions pour nier les contraintes, mais pour le reste on ne considère que des rationnels. Bref, sur les expressions, on type en entier, mais sur les polyèdres, on type en réel.

55. **Extraction de propriétés**

⟨*abstract0.h*⟩+≡

```
/* ============================================================ */
/* II.4 Extraction of properties */
/* ============================================================ */

interval_t abstract_bound_linexpr(manager_t* man,
                                  abstract_t* a, linexpr_t* expr);
  /* Returns the interval taken by a linear expression
     over the abstract value */

lincons_t** abstract_to_lincons_array(manager_t* man, abstract_t* a, int* size);
  /* Converts an abstract value to a polyhedra
     (conjunction of linear constraints).
     The size of the returned array is stored in size. */

boundinterval_t* abstract_to_box(manager_t* man, abstract_t* a);
  /* Converts an abstract value to an interval/hypercube.
     The size of the resulting array is abstract_dimension(man,a).  This
     function can be reimplemented by using abstract_bound_linexpr */
```

5.3 Opérations (version fonctionnelle

56. Bornes supérieures et inférieures

⟨*abstract0.h*⟩+≡

```
  /* ********************************************************************** */
  /* III. Operations : functional version */
  /* ********************************************************************** */

  /* ============================================================ */
  /* III.1 Meet and Join */
  /* ============================================================ */

  abstract_t* abstract_meet(manager_t* man, abstract_t* a1, abstract_t* a2);
  abstract_t* abstract_join(manager_t* man, abstract_t* a1, abstract_t* a2);
    /* Meet and Join of 2 abstract values */

  abstract_t* abstract_meet_array(manager_t* man, abstract_t** tab, int size);
  abstract_t* abstract_join_array(manager_t* man, abstract_t** tab, int size);
    /* Meet and Join of an array of abstract values */
       QUESTION : should we impose that size>0? If size=0, it is not possible to
       define the dimensionality of the result */

  abstract_t* abstract_meet_lincons(manager_t* man,
                                    abstract_t* a, lincons_t* c);
  abstract_t* abstract_meet_intlincons(manager_t* man,
                                       abstract_t* a, intlincons_t* c);
    /* Meet of an abstract value with a constraint */

  abstract_t* abstract_meet_lincons_array(manager_t* man,
                                          abstract_t* a,
                                          lincons_t** tlincons, int size);
    /* Meet of an abstract value with a set of constraints
       (generalize abstract_of_lincons_array) */

  abstract_t* abstract_add_ray_array(manager_t* man,
                                     abstract_t* a,
                                     ray_t** tray, int size);
    /* Generalized time elapse operator */
```

Discussion sur `abstract_meet_lincons_array` :
- Plus général que `abstract_of_lincons_array`, *c.f.* §5.2.
- Même si implanté par appel successifs à `abstract_meet_lincons`, les appels intermédiaires peuvent utiliser des effets de bord.
- Dans le cas des polyèdres, plus efficace que des appels successifs, et plus efficace qu'appeler `abstract_of_lincons_array` et faire l'intersection.

A-t-on besoin d'une version *array* de `abstract_meet_intlincons` ? ([bj] Je ne le pense pas).

57. **Affectations et Substitutions.**

Il a été décidé de ne pas implanter les *weak update*. D'après Antoine Miné, cela ne va pas assez loin. D'apères Bertrand Jeannet,, Reps, Sagiv & al ne l'utilise pas. Les *weak update* peuvent par ailleurs être implantés à partir des opérations ci-dessus. En revanche, l'expansion et le pliage de dimension est fourni pour autoriser la notion de *weak update* à un niveau supérieur, voir la suite.

⟨*abstract0.h*⟩+≡
```
  /* ============================================================ */
  /* III.2 Assignement and Substitutions */
  /* ============================================================ */

  abstract_t* abstract_assign_linexpr(manager_t* man,
                                      abstract_t* a,
                                      dim_t dim, linexpr_t* expr);
  abstract_t* abstract_substitute_linexpr(manager_t* man,
                                          abstract_t* a,
                                          dim_t dim, linexpr_t* expr);
  abstract_t* abstract_assign_intlinexpr(manager_t* man,
                                         abstract_t* a,
                                         dim_t dim, intlinexpr_t* expr);
  abstract_t* abstract_substitute_intlinexpr(manager_t* man,
                                             abstract_t* a,
                                             dim_t dim, intlinexpr_t* expr);
    /* Assignement and Substitution of a single dimension by resp.
       a linear expression and a interval linear expression */

  abstract_t* abstract_assign_linexpr_array(manager_t* man,
                                            abstract_t* a,
                                            dim_t* tdim,
                                            linexpr_t** texpr,
                                            int size);
  abstract_t* abstract_substitute_linexpr_array(manager_t* man,
                                                abstract_t* a,
                                                dim_t* tdim,
                                                linexpr_t* texpr,
                                                int size);
    /* Parallel Assignement and Substitution of several dimensions by
       linear expressons. */
```

58. **Projections et Quantifications existentielles.**

⟨*abstract0.h*⟩+≡

```
/* ============================================================ */
/* III.3 Projections */
/* ============================================================ */

abstract_t* abstract_project(manager_t* man, abstract_t* a, dim_t dim);
abstract_t* abstract_forget(manager_t* man, abstract_t* a, dim_t dim);
abstract_t* abstract_project_array(manager_t* man,
                                   abstract_t* a, dim_t* tdim, int size);
abstract_t* abstract_forget_array(manager_t* man,
                                  abstract_t* a, dim_t tdim, int size);
```

Questions :

- Peut-on distinguer `projet` de `forget` juste avec un flag histoire de diminuer le nombre de fonctions ? + voir remarque suivante.
- Dans tous les treillis envisagés, `project` peut s'implanter à partir de `forget` suivi d'une affectation à zéro. La réciproque est vraie, en utilisant `add_ray_array`, mais c'est sans doute plus compliqué pour pas mal de treillis.
- Nommage de `forget` : on peut aussi l'appeler `exist`. De manière générale, doit-on utiliser un vocabulaire mathématique (`exist`, `least upper bound`, ...) ou le vocabulaire d'usage, notamment outre-atlantique (`forget`, `join`, ...) ?

59. **Changement et Permutations de dimensions.**

On fournit à la fois des fonctions d'usage simple, pour ajouter ou retirer des dimensions "à la fin" des dimensions existantes, et des fonctions plus puissantes, permettant d'effectuer des permutations et nécessaires pour le niveau 1 de l'interface.

Question identique à celle du paragraphe précédent : utilise-t-on un flag pour distinguer `project` de `forget` ou `embed` ?

⟨*abstract0.h*⟩+≡

```
/* ========================================================== */
/* III.4 Change and permutation of dimensions */
/* ========================================================== */

/* ------------------------------ */
/* III.4a Add at the end */
/* ------------------------------ */

abstract_t* abstract_add_dimensions_and_project(manager_t* man,
                                                abstract_t* a,
                                                int intdim, int realdim);
abstract_t* abstract_add_dimensions_and_forget(manager_t* man,
                                               abstract_t* a,
                                               int intdim, int realdim);
  /* Adds intdim (resp. realdim) dimensions at the end of integer (resp. real)
     variables range */

abstract_t* abstract_remove_dimensions(manager_t* man,
                                       abstract_t* a, int intdim, int realdim);
  /* Removes intdim (resp. realdim) dimensions at the end of integer
     (resp. real) variables range. Implicitly project (or forget)
     the removed dimensions. */
```

⟨*abstract0.h*⟩+≡

```
  /* ------------------------------ */
  /* III.4b Add anywhere and permute */
  /* ------------------------------ */

  /* Provides necessary stuff for level 1 to automatically adjust abstract values
     defined on different variables, e.g. A1(x,y,z) inter A2(z,y,w) */

  abstract_t* abstract_add_permute_dim_and_forget(manager_t* man,
                                                  abstract_t* a,
                                                  int intdim, int realdim,
                                                  int* permutation);
  abstract_t* abstract_add_permute_dim_and_project(manager_t* man,
                                                   abstract_t* a,
                                                   int intdim, int realdim,
                                                   int* permutation);
    /* Adds intdim (resp. realdim) dimensions at the end of integer (resp. real)
       variables range, and then permutes dimensions with i -> permutation[i].
       Assumption :
       1. permutation is of size the dimension on the resulting abstract value
       2. integer dimensions should remain before real dimensions.

       Example : if we have resp. a vertex  [b0,b1,b2,b3] and a constraint
       a0 x0 + a1 x1 + a2 x2 + a3 x3 >= 0 in an abstract value a, to which
       is applied

       abstract_add_permute_dim_and_project(man,a, 0, 3, [0,4,1,2,5,6,3]),

       the vertecs becomes [b0,0,b1,b2,0,0,b3] and the constraint becomes
       a0 x0 + a1 x2 + a2 x3 + a3 x7 >= 0.

       Conversely, if we apply

       abstract_permute_remove_dim(man,a, 0, 2, [2,0,3,1]),

       the vertecs becomes [b1,b3].
    */

  abstract_t* abstract_permute_remove_dim(manager_t* man,
                                          abstract_t* a,
                                          int intdim, int realdim,
                                          int* permutation);
    /* Permutes the dimensions with i -> permutation[i], and then
       remove the last dimensions */
```

60. **Expansion et pliage**

Il a été décidé de fournir au niveau 0 un support pour l'expansion et le pliage de dimensions.

Formellement, la sémantique concrète est la suivante (sachant que ces opérations sont associatives, et que z est expansé en z et w, puis z et w plié sur z) :

$$\mathsf{expand}(P(x,y,z),z,w) = P(x,y,z) \wedge P(x,y,w) \tag{5.1}$$

$$\mathsf{fold}((Q(x,y,z,w),z,w) = (\exists w : Q(x,y,z,w)) \vee (\exists z : Q(x,y,z,w))[z \leftarrow w] \tag{5.2}$$

Là encore, ces opérations ne sont pas minimales, mais pour des raisons de performances il semble que ça vaille le coup de fournir des opérations spécifiques.

⟨*abstract0.h*⟩+≡

```
/* ========================================================== */
/* III.5 Expansion and folding of dimensions */
/* ========================================================== */

abstract_t* abstract_expand_dim(manager_t* man,
                                abstract_t* a,
                                dim_t dim,
                                int n);
  /* Expand the dimension dim into itself + n additional dimensions.
     It results in (n+1) unrelated dimensions having same
     relations with other dimensions. The (n+1) dimensions are put as follows :

     - original dimension dim

     - if the dimension is integer, the n additional dimensions are put at the
       end of integer dimensions; if it is real, at the end of the real
       dimensions.
  */

abstract_t* abstract_fold_dim(manager_t* man,
                              abstract_t* a,
                              dim_t* tdim,
                              int size);
  /* Fold the dimensions in the array tdim of size n>=1 and put the result
     in the first dimension in the array. The other dimensions of the array
     are then removed (using abstract_permute_remove_dimensions). */
```

61. **Élargissement.**

⟨*abstract0.h*⟩+≡

```
/* ========================================================== */
/* III.6 Widening */
/* ========================================================== */

/* Widening with threshold */
abstract_t* abstract_widening(manager_t* man,
                              abstract_t* a1, abstract_t* a2,
                              lincons_t** tab, int size);
```

62. **Clôture topologique.**

Il faut décider si l'on traite les contraintes strictes. [bj] Ça ne mange pas de pain, il reste toujours la possibilité d'ignorer le caractère stricte d'une contrainte pour une implantation particulière.

⟨*abstract0.h*⟩+≡

```
/* ============================================================ */
/* III.7 Closure operation */
/* ============================================================ */

/* Returns the topological closure of a possibly opened abstract value */

abstract_t* abstract_closure(manager_t* man, abstract_t* a);
```