Farkas calculator

A short documentation

Christophe Alias

1 Syntax

prog: (`parameters = {` *parameter*`, ..., ` *parameter* `};`)?
instruction ; ... ; **instruction**

instruction:
`object`
Display the object (`iscc format for the polyhedra`*)*
| `ID :=` object
symbol_table[ID] := object
| `lexmin` object
Display the lexicographic minimum of the object (expect a polyhedron obtained with `keep`*)*
| `set ID`
Display "`ID := `*", for iscc scripting*
| `comment ID`
Display "`# ID`*" and a carriage return, for iscc scripting*

object:
`ID`
symbol_table[ID]
| polyhedron
| affine_form
| affine_function

polyhedron:
`ID`
symbol_table[ID] (expect a polyhedron)
| `[ ] -> { [` *variable*`, ..., ` *variable* `] :` *inequation* `and ... and` *inequation* `}`
iscc-compliant definition of a polyhedron
| `solve` affine_form `= 0`
Farkas identification on affine_form (expected in Farkas form)

| `define` affine_form `with ID`

Express the coefficients of affine_form (named ID_k) in terms of lambdas (expect affine_form in Farkas form)

| `find` ID_1, ..., ID_n `s.t.` affine_form `= 0`

Farkas identification + projection on the coefficients for the affine forms assigned to symbols ID_k. This command has the same effect as projecting on $ID_1, ..., ID_n$ the polyhedron (`solve` *affine_form = 0)**(`define` ID_1 `with` ID_1) `*` ... `*` (`define` ID_n `with` ID_n).*

| `keep` *variable*`, ...,` *variable* `in` polyhedron

Project the polyhedron on the variables, keep the variable order for lexmin. By commodity, parameters are allowed among variables. Each parameter `p` *among the variable list is turned to the variable* `p_counter`*.*

| polyhedron `*` ... `*` polyhedron

Set intersection

inequation:

`true`

| `false`

| expression [`>`,`<`,`>=`,`<=`] expression

expression:

expression [`+`,`-`] expression

| `INT` * expression

| expression * `INT`

leaf_affine_form:

`ID`

symbol_table[ID], expect an affine form

| { `[` *variable*, ..., *variable* `]` `->` expression }

Build an affine form. Parameters are allowed and handled as constants.

| `positive_on` polyhedron

Affine form positive on the polyhedron in Farkas form (expect a non-parametrized polyhedron).

| leaf_affine_form `.` affine_function

Composition affine form ∘ affine function.

| `INT` `*` leaf_affine_form

Scaling

affine_form:

leaf_affine_form

| leaf_affine_form [+,-] ... [+,-] leaf_affine_form

Function addition, expect the same input dimension. Constants are possible (then interpreted as affine forms).

affine_function:

ID

symbol_table[ID] (expect an affine function)

| { [*variable*, ..., *variable*] -> [expression , ... , expression] }

Build an affine function (expect non-parametrized expressions).

2 Examples

Affine scheduling with direct dependences

File `poly.fk`, launch with `fk poly.fk`.

```
#
# Polynomial product, direct dependences
#

iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
theta := positive_on iterations;

dependence := {[i,j,N]->[i - 1,j+1,N]};
D := find theta s.t. theta - (theta . dependence) - 1 = 0;

#Display the schedule domain (iscc scripting)
set schedule_domain;
D;

#Pick a solution
lexmin (keep theta_0,theta_1,theta_2,theta_3 in D)
```

Affine scheduling with relational dependences (PRDG)

File `poly2.fk`

```
#
# Polynomial product, PRDG dependences
#
```

```
iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
theta := positive_on iterations;

dependence := [] -> { [is,js,id,jd,N]: 0 <= is and is <= N  and 0 <= js and js <= N
                      and 0 <= id and id <= N  and 0 <= jd and jd <= N
                      and is+js = id+jd and is<id};
causality := positive_on dependence;
to_target := {[is,js,id,jd,N]->[id,jd,N]};
to_source := {[is,js,id,jd,N]->[is,js,N]};

D := find theta s.t. (theta . to_target) - (theta . to_source) - causality - 1 = 0;

#Pick a solution
lexmin (keep theta_0,theta_1,theta_2,theta_3 in D)
```

Affine scheduling with latency minimization

File `poly3.fk`

```
#
# Polynomial product with latency minimization
#

iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
theta := positive_on iterations;

dependence := [] -> { [is,js,id,jd,N]: 0 <= is and is <= N  and 0 <= js and js <= N
                      and 0 <= id and id <= N  and 0 <= jd and jd <= N
                      and is+js = id+jd and is<id};

#
# Correctness: s -> t ==> theta(s) < theta(t)
#
causality := positive_on dependence;
to_target := {[is,js,id,jd,N]->[id,jd,N]};
to_source := {[is,js,id,jd,N]->[is,js,N]};
theta_correct := solve (theta . to_target) - (theta . to_source) - causality - 1 = 0;
theta_def := define theta with theta;

#
# Efficiency: theta(s) <= latency(N), then min latency(N)
```

```
#

# L(N) >= 0 on the parameter domain
latency := positive_on ([] -> {[N]: N >= 0});

# theta(i) <= L(N) \forall i,N
bound_theta := positive_on ([] -> {[i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N});
theta_bounded := solve (latency . {[i,j,N] -> [N]}) - theta- bound_theta = 0;
bound_def := define latency with latency;

# Display the result
lexmin (
  keep latency_0,latency_1,theta_0,theta_1,theta_2,theta_3
  in theta_correct*theta_def*theta_bounded*bound_def
)
```

Affine scheduling with dependence selection

File `poly4.fk`

```
#
# Polynomial product, parametrized delay
#

parameters := {eps,inv_eps};

iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
theta := positive_on iterations;

dependence := [] -> { [is,js,id,jd,N]: 0 <= is and is <= N  and 0 <= js and js <= N
                      and 0 <= id and id <= N  and 0 <= jd and jd <= N
                      and is+js = id+jd and is<id};

#
# Correctness: s -> t ==> theta(s) <= theta(t) + eps, 0 <= eps <= 1
#
causality := positive_on dependence;
to_target := {[is,js,id,jd,N]->[id,jd,N]};
to_source := {[is,js,id,jd,N]->[is,js,N]};
theta_correct := solve (theta . to_target) - (theta . to_source) - causality
                        + {[is,js,id,jd,N] -> -1*eps} = 0;
theta_def := define theta with theta;
```

```
eps_correct := [] -> {[i]: 0 <= eps and eps <= 1 and inv_eps = 1-eps};

# Display the result
lexmin (
  keep inv_eps,theta_0,theta_1,theta_2,theta_3
  in theta_correct*theta_def*eps_correct;
)
```

Affine scheduling with latency minimization and dependence selection

File `poly5.fk`

```
#
# Polynomial product, final
#

parameters := {eps,inv_eps};

iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
theta := positive_on iterations;

dependence := [] -> { [is,js,id,jd,N]: 0 <= is and is <= N  and 0 <= js and js <= N
                      and 0 <= id and id <= N  and 0 <= jd and jd <= N
                      and is+js = id+jd and is<id};

#
# Correctness: s -> t ==> theta(s) <= theta(t) + eps, 0 <= eps <= 1
#
causality := positive_on dependence;
to_target := {[is,js,id,jd,N]->[id,jd,N]};
to_source := {[is,js,id,jd,N]->[is,js,N]};
theta_correct := solve (theta . to_target) - (theta . to_source) - causality
                        + {[is,js,id,jd,N] -> -1*eps} = 0;
theta_def := define theta with theta;
eps_correct := [] -> {[i]: 0 <= eps and eps <= 1 and inv_eps = 1-eps};

#
# Efficiency: theta(s) <= latency(N), then min latency(N)
#
```

```
# L(N) >= 0 on the parameter domain
latency := positive_on ([] -> {[N]: N >= 0});

# theta(i) <= L(N) \forall i,N
bound_theta := positive_on ([] -> {[i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N});
theta_bounded := solve (latency . {[i,j,N] -> [N]}) - theta- bound_theta = 0;
bound_def := define latency with latency;


# Display the result
lexmin (
  keep inv_eps,latency_0,latency_1,theta_0,theta_1,theta_2,theta_3,eps
  in theta_correct*theta_def*eps_correct*theta_bounded*bound_def;
)
```