`fkcc`: the Farkas Calculator

Christophe Alias

CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
`Christophe.Alias@ens-lyon.fr`
`http://foobar.ens-lyon.fr/fkcc`

Abstract. In this paper, we present FKCC, a scripting tool to prototype program analyses and transformations exploiting the affine form of Farkas lemma. Our language is general enough to prototype in a few lines sophisticated termination and scheduling algorithms. The tool is freely available and may be tried online via a web interface. We believe that FKCC is the missing chain to accelerate the development of program analyses and transformations exploiting the affine form of Farkas lemma.

Keywords: Farkas lemma · Scripting tool · Termination · Scheduling

1 Introduction

Many program analyses and transformations require to handle conjunction of affine constraints C and C' with a universal quantification as $\forall x : x \models C \Rightarrow x \models C'$. For instance, this appears in loop scheduling [6, 7], loop tiling [2], program termination [1] and generation of invariants [3]. Farkas lemma – affine form – provides a way to get rid of that universal quantification, at the price of introducing quadratic terms. In the context of program termination and loop scheduling, it is even possible to use Farkas lemma to turn universally quantified quadratic constraints into *existentially quantified affine constraints*. This requires tricky algebraic manipulations, not easy to applied by hand, neither to implement.

In this paper, we propose a scripting tool, FKCC, which makes it possible to manipulate easily Farkas lemma to benefit from those nice properties. More specifically, we made the following contributions:

– A general formulation for the resolution of equations $\forall x : S(\boldsymbol{x}) = 0$ where S is summation of affine forms including Farkas terms. So far, this resolution was applied for specific instances of Farkas summation. This result is the basic engine of the FKCC scripting language.
– A scripting language to apply and exploit Farkas lemma; among polyhedra, affine functions and affine forms.
– Our tool, FKCC, implementing these principles, available at http://foobar.ens-lyon.fr/fkcc. FKCC may be downloaded and tried online *via* a web interface. FKCC comes with many examples, making it possible to adopt the tool easily.

This paper is structured as follows. Section 2 presents the affine form of Farkas lemma, our resolution theorem, and explains how it applies to compute scheduling functions. Then, Section 3 defines the syntax and outlines informally the semantics of the FKCC language. Section 4 presents two complete use-cases of FKCC. Finally, Section 5 concludes this paper and draws future research perspectives.

2 Farkas Lemma in Program Analysis and Compilation

This section presents the theoretical background of this paper. We first introduce the affine form of Farkas lemma. Then, we present our theorem to solve equations $S(\boldsymbol{x}) = 0$ where S is a summation of affine forms including Farkas terms. This formalization will then be exploited to design the FKCC language.

Lemma 1 (Farkas Lemma, affine form). *Consider a convex polyhedron* $\mathcal{P} = \{\boldsymbol{x},\ A\boldsymbol{x} + \boldsymbol{b} \geq 0\} \subseteq \mathbb{R}^n$ *and an affine form* $\phi : \mathbb{R}^n \to \mathbb{R}$ *such that* $\phi(\boldsymbol{x}) \geq 0$ $\forall \boldsymbol{x} \in \mathcal{P}$.
Then: $\exists \boldsymbol{\lambda} \geq \mathbf{0}, \lambda_0 \geq 0$ *such that:*

$$\phi(\boldsymbol{x}) = {}^t\boldsymbol{\lambda}(A\boldsymbol{x} + \boldsymbol{b}) + \lambda_0 \quad \forall \boldsymbol{x}$$

Hence, Farkas lemma makes it possible to remove the quantification $\forall \boldsymbol{x} \in \mathcal{P}$ by encoding directly the positivity over $\mathcal{P}$ into the definition of ϕ, thanks to the Farkas multipliers $\boldsymbol{\lambda}$ and λ_0. In the remainder, *Farkas terms* will be denoted by: $\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(\boldsymbol{x}) = {}^t\boldsymbol{\lambda}(A\boldsymbol{x} + \boldsymbol{b}) + \lambda_0$. We now propose a theorem to solve equations $S(\boldsymbol{x}) = 0$ where S involves Farkas terms. The result is expressed as a conjunction of affine constraints, which is suited for integer linear programming:

Theorem 1. *Consider a summation* $S(\boldsymbol{x}) = \boldsymbol{u} \cdot \boldsymbol{x} + v + \sum_i \mathfrak{F}(\lambda_{i0}, \boldsymbol{\lambda_i}, A_i, \boldsymbol{b}_i)(\boldsymbol{x})$ *of affine forms, including Farkas terms. Then:*

$$\forall \boldsymbol{x} :\ S(\boldsymbol{x}) = 0 \quad \textit{iff} \quad \begin{cases} \boldsymbol{u} + \sum_i {}^tA_i\boldsymbol{\lambda}_i = \mathbf{0} \ \wedge \\ v + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) = 0 \end{cases}$$

Proof. We have:

$$\begin{aligned} S(\boldsymbol{x}) &= {}^t\boldsymbol{x}\left(\sum_i {}^tA_i\boldsymbol{\lambda}_i\right) + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) + \boldsymbol{u} \cdot \boldsymbol{x} + v \\ &= {}^t\boldsymbol{x}\left(\boldsymbol{u} + \sum_i {}^tA_i\boldsymbol{\lambda}_i\right) + v + \sum_i (\boldsymbol{\lambda}_i \cdot \boldsymbol{b}_i + \lambda_{0i}) \end{aligned}$$

$S(\boldsymbol{x}) = \boldsymbol{\tau} \cdot \boldsymbol{x} + \tau_0 = 0$ for any $\boldsymbol{x}$ iff $\boldsymbol{\tau} = \mathbf{0}$ and $\tau_0 = 0$. Hence the result. $\square$

Application to scheduling Consider the polynomial product kernel depicted in Figure 3.(a). Farkas lemma and Theorem 1 may be applied to compute a *schedule*, this is a way to reorganize the computation of the program to fulfill various criteria (overall latency, locality, parallelism, etc). On this example, a schedule may be expressed as an *affine form* $\theta : (i, j) \mapsto t$ assigning a *timestamp*

$t \in \mathbb{Z}$ to each iteration (i,j). This way, a schedule *prescribes* an execution order $\prec_\theta := \{((i,j),(i',j')) \mid \theta(i,j) < \theta(i',j')\}$. Figure 3.(b) illustrates the order prescribed by the schedule $\theta(i,j) = i$: a sequence of vertical wave fronts, whose iterations are executed in parallel.

A schedule must be positive everywhere on the set of *iteration vectors* $\mathcal{D}_N = \{(i,j) \mid A\ ^t(i,j,N) + \boldsymbol{b}\}$ (referred to as *iteration domain*). In general, the iterations domains are parametrized (typically by the array size N) and the schedule may depends on N. Hence we have to consider vectors (i,j,N) instead of (i,j):

$$\theta(i,j,N) \geq 0 \quad \forall (i,j) \in \mathcal{D}_N \tag{1}$$

Applying Farkas lemma, this translates to:

$$\exists \lambda_0 \geq 0, \boldsymbol{\lambda} \geq 0 \quad \text{such that} \quad \theta(i,j,N) = \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i,j,N) \tag{2}$$

Moreover, a schedule must *satisfy the data dependencies* $(i,j) \to (i',j')$. $\to$ is generally expressed as a Presburger relation [8], in turned *abstracted* as a rational convex polyhedron Δ_N containing the correct vectors (i,j,i',j') and sometimes false positives. Here again, $\Delta_N = \{(i,j,i',j') \mid C\ ^t(i,j,i',j',N) + \boldsymbol{d} \geq 0\}$ is parametrized by structure parameter N. This way, the correctness condition translates to:

$$\theta(i',j',N) > \theta(i,j,N) \quad \forall (i,j,i',j') \in \Delta_N \tag{3}$$

Note that $\theta(i',j',N) > \theta(i,j,N)$ is equivalently written as the positivity of an affine form over a convex polyhedron: $\theta(i',j',N) - \theta(i,j,N) - 1 \geq 0$. Applying Farkas lemma:

$$\exists \mu_0 \geq 0, \boldsymbol{\mu} \geq 0 \text{ such that } \theta(i',j',N) - \theta(i,j,N) - 1 = \mathfrak{F}(\mu_0, \boldsymbol{\mu}, C, \boldsymbol{d})(i,j,i',j',N)$$

Substituting θ using Equation (2), this translates to $S(i,j,i',j',N) = 0$, where $S(i,j,i',j',N)$ is defined as the summation:

$$\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i',j',N) - \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(i,j,N) - \mathfrak{F}(\mu_0, \boldsymbol{\mu}, C, \boldsymbol{d})(i,j,i',j',N) - 1$$

Since $-\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b}) = \mathfrak{F}(-\lambda_0, -\boldsymbol{\lambda}, A, \boldsymbol{b})$, we may apply theorem 1 to obtain a system of affine constraints with $\lambda_0, \boldsymbol{\lambda}, \mu_0, \boldsymbol{\mu}$. Linear programming may then be applied to find out the desired schedule [2, 7]. The same principle might be applied in *termination analysis* to derive a ranking function [1], this will be developed in Section 4.

3 Language

This section specifies the input language of FKCC and outlines informally its semantics. Figure 1 depicts the input syntax of FKCC. Keywords and syntax sugar are written with `verbatim` letters, identifiers with *italic* letter and syntactic categories with roman letters. Among identifiers, p is a parameter, v is a variable (typically a loop counter) and *id* is an FKCC identifier.

```
program ::= (parameters = { p, ..., p };)?  instruction; ...; instruction;

instruction ::= object | id := object | lexmin polyhedron | lexmax polyhedron | set id

object ::= polyhedron | affine_form | affine_function

polyhedron ::=
  [ p, ..., p ] -> { [ v, ..., v ] : inequation and ... and inequation }
| polyhedron * ... * polyhedron
| solve affine_form = 0
| define affine_form with v
| keep v, ..., v in polyhedron
| find id, ..., id  s.t. affine_form = 0

affine_form ::= leaf_affine_form | leaf_affine_form [+-] ... [+-] leaf_affine_form

leaf_affine_form ::=
  { [ v, ..., v ] -> expression }
| positive_on polyhedron
| leaf_affine_form . affine_function
| int
| int * leaf_affine_form

affine_function ::= { [ v, ..., v ] -> [ expression, ..., expression ] }
```

Fig. 1. `fkcc` syntax

Program, instructions, polyhedra An FKCC program consists of a sequence of instructions. There is no other control structure than the sequence. An instruction may assign an FKCC object (polyhedron, affine form or affine function) to an FKCC identifier, or may be an FKCC object alone. In the latter, the FKCC object is streamed out to the standard output. Also, we often need to compute the lexicographic optimum of a polyhedron, typically to pick an optimal schedule. FKCC uses *parameteric integer linear programming* [5] *via* the Piplib library. The result is a discussion on the parameter value:

```
parameters := {N};
lexmin [N] -> {[i,j]: 0 <= i and i <= N and 0 <= j and j <= N};
```

would give:

```
if(N >= 0)
  {
    [] -> {[0,0]}
  }
else
  {
    (no solution)
  }
;
```

Note that structure parameters *must* be declared with the `parameters` construct. When no parameters are involved, the `parameters` construct may be omitted. To ensure the compatibility with ISCC [10] syntax, the parameters of a polyhedron *may* be declared on preceding brackets `[N] -> ...`. This is purely optional: FKCC actually does not analyze this part. The instruction `set` *id* emits `id :=` to the standard output. This makes it possible to generate ISCC scripts for further analysis. Finally, the set intersection of two polyhedra `P` and `Q` is obtained with `P*Q`.

Affine forms An affine form may be defined as a *Farkas term*:

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

If `iterations` is $\{\boldsymbol{x} \mid A\boldsymbol{x} + \boldsymbol{b} \geq 0\}$, then `theta` is defined as $\mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})$ where λ_0 and $\boldsymbol{\lambda}$ are fresh positive variables. In that case, the polyhedron is *never* parametrized: the *parameters must be handled as variables.* In particular, do not name variables with identifiers declared as parameters with `parameters :=`, as they would be treated as parameters whatever the context. Affine forms might be summed, scaled and composed with *affine functions*, typically to adjust the input dimension:

```
to_target := {[i,j,i',j',N]->[i,j,N]};
to_source := {[i,j,i',j',N]->[i',j',N]};
sum := theta.to_target - 2*theta.to_source + 1 + {[i,j,i',j',N] -> 2*i-i'};
```

In a summation of affine forms, affine forms must have the same input dimension. Also, a constant (`1`) is automatically interpreted as an affine form (`[i,j,i',j',N] -> 1`). Affine forms may also be stated explicitely (`{[i,j,i',j',N] -> 2*i-i'}`). The terms of the summation are simply separated with `+` and `-`, no parenthesis are allowed.

Resolution The main feature of FKCC is the resolution of equations $S(\boldsymbol{x}) = 0$ where S is a summation of affine forms including Farkas terms. This is obtained with the instruction `solve`:

```
solve sum = 0;
```

The result is a polyhedron with Farkas multipliers (obtained after applying Theorem 1):

```
[] -> {[lambda_0,lambda_1,lambda_2,lambda_3,lambda_4] :
        (2+lambda_0)+(-1*lambda_1) >= 0 and (-2+(-1*lambda_0))+lambda_1 >= 0 and
        lambda_2+(-1*lambda_3) >= 0 and (-1*lambda_2)+lambda_3 >= 0 and
        (-1*lambda_1)+(-1*lambda_3) >= 0 and lambda_1+lambda_3 >= 0 and
        (-1+(-2*lambda_0))+(2*lambda_1) >= 0 and (1+(2*lambda_0))+(-2*lambda_1) >= 0 and
        (-2*lambda_2)+(2*lambda_3) >= 0 and (2*lambda_2)+(-2*lambda_3) >= 0 and
        1+(-1*lambda_4) >= 0 and -1+lambda_4 >= 0 and lambda_4 >= 0 and
        lambda_0 >= 0 and lambda_1 >= 0 and lambda_2 >= 0 and lambda_3 >= 0 and
        lambda_4 >= 0 and lambda_0 >= 0 and lambda_1 >= 0 and lambda_2 >= 0 and lambda_3 >= 0};
```

At this point, we need to recover the coefficients of our affine form `theta` in terms of $\boldsymbol{\lambda}$ (`lambda_0,lambda_1,lambda_2,lambda_3`) and λ_0 (`lambda_4`). Observe that `theta`$(\boldsymbol{x}) = \mathfrak{F}(\lambda_0, \boldsymbol{\lambda}, A, \boldsymbol{b})(\boldsymbol{x}) = {}^t\boldsymbol{\lambda}A\boldsymbol{x} + \boldsymbol{\lambda} \cdot \boldsymbol{b} + \lambda_0$. If the coefficients of `theta` are written: `theta`$(\boldsymbol{x}) = \boldsymbol{\tau} \cdot \boldsymbol{x} + \tau_0$, we simply have: $\boldsymbol{\tau} = {}^t\boldsymbol{\lambda}A$ and $\tau_0 = \boldsymbol{\lambda} \cdot \boldsymbol{b} + \lambda_0$. This is obtained with `define`:

```
define theta with tau;
```

The result is a conjunction of definition equalities, gathered in a polyhedron:

```
[] -> {[lambda_0,lambda_1,lambda_2,lambda_3,lambda_4,tau_0,tau_1,tau_2,tau_3] :
        ((-1*lambda_0)+lambda_1)+tau_0 >= 0 and (lambda_0+(-1*lambda_1))+(-1*tau_0) >= 0 and
        ((-1*lambda_2)+lambda_3)+tau_1 >= 0 and (lambda_2+(-1*lambda_3))+(-1*tau_1) >= 0 and
        ((-1*lambda_1)+(-1*lambda_3))+tau_2 >= 0 and (lambda_1+lambda_3)+(-1*tau_2) >= 0 and
        (-1*lambda_4)+tau_3 >= 0 and lambda_4+(-1*tau_3) >= 0};
```

The first coefficients `tau_k` define $\boldsymbol{\tau}$, the last one defines the constant τ_0. On our example, `theta(i,j,N) = tau_0*i + tau_1*j + tau_2*N + tau_3`. Now we may gather the results and eliminate the λ to keep only $\boldsymbol{\tau}$ and τ_0:

```
keep tau_0,tau_1,tau_2,tau_3 in ((solve sum = 0)*(define theta with tau));
```

The result is a polyhedron with the solutions. Here, there are no solutions: the result is an empty polyhedron. All these steps may be applied once with the `find` command:

```
find theta s.t. sum = 0;
```

The coefficients are automatically named `theta_0`, `theta_1`, etc with the same convention as `define`. We point out that `define` *choose fresh names* for coefficients (e.g. `tau_4`, `tau_5` on the second time with ``tau'') whereas `find` *always choose the same names.* Hence `find` would be prefered when deriving separately constraints on the same coefficients of `theta`. `find` may filter the coefficients for several affine forms expressed as Farkas terms in a summation:

```
find theta_S,theta_T s.t.
  theta_T.to_target - theta_S.to_source - 1
     - (positive_on dependences_from_S_to_T) = 0;
```

This is typically used to compute schedules for programs with multiple assignments (here S and T with dependences from iterations of S to iterations of T). Finally, note that `keep tau_0,tau_1,tau_2,tau_3 in P;` projects `P` on variables `tau_0,tau_1,tau_2,tau_3`: the result is a polyhedron with integral points of coordinates `(tau_0,tau_1,tau_2,tau_3)`. This way, the order in which `tau_0,tau_1,tau_2,tau_3` are specified to `keep` impacts directly a further lexicographic optimization.

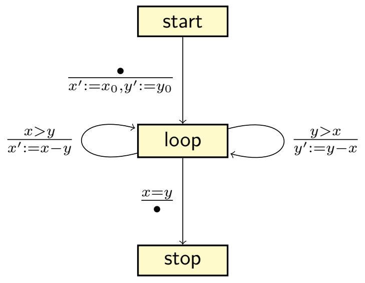
4 Examples

This section shows how FKCC might be used to specify in a few lines termination analysis and loop scheduling.

4.1 Termination analysis

```
assert $x_0 > 0 \wedge y_0 > 0$
$x := x_0$; $y := y_0$;
while $x \neq y$
  if $x > y$
  then $x := x - y$;
  else $y := y - x$;
```

(a) Kernel



(b) Affine automaton

$$\mathcal{I}_{\text{start}} = \{x_0 > 0 \wedge y_0 > 0\}$$
$$\mathcal{I}_{\text{loop}} = \{x > 0 \wedge y > 0 \wedge x \leq x_0 \wedge y \leq y_0\}$$
$$\mathcal{I}_{\text{stop}} = \{x \leq y_0 \wedge x \leq x_0 \wedge x > 0 \wedge x = y\}$$

$$\rho_{\text{start}}(x, y) = (2)$$
$$\rho_{\text{loop}}(x, y) = (1, x + y - 2)$$
$$\rho_{\text{stop}}(x, y) = (0)$$

(c) Invariants and ranking

Fig. 2. Termination example

Consider the example depicted on Figure 2. The program computes the gcd of two integers x_0 and y_0 (a). It is translated to an affine automaton (b) (also called integer interpreted automaton), in turn analyzed to check the termination (c): does the program terminates for *any* input (x_0, y_0) satisfying the precondition $x_0 > 0 \wedge y_0 > 0$?

This problem is – as most topics in static analysis – undecidable in general. However, we may conclude when it is possible to derive statically an abstraction precise enough of the program execution. In [1], we provide a termination algorithm based on the computation of a *ranking*. A ranking is an application $\rho_{\text{label}} : \mathbb{Z}^n \to (\mathcal{R}, \prec)$ which maps each reachable state of the automaton

$\langle label, \boldsymbol{x} \rangle$ to a *rank* belonging to well-founded set. On our example a reachable state could be $\langle loop, (x:3, y:3, x_0:3, y_0:6) \rangle$ after firing the initial transition and the right transition.

The ranking is decreasing on the transitions: for any transition $\langle label, \boldsymbol{x} \rangle \to \langle label', \boldsymbol{x}' \rangle$, we have: $\rho_{\text{label'}}(\boldsymbol{x}') \prec \rho_{\text{label}}(\boldsymbol{x})$. Since ranks belong to a well founded set, there are – by definition – no infinite decreasing chain of ranks. Hence infinite chains of transitions from an initial state never happen.

On [1], we provide a general method for computing a ranking of an affine automaton. Our ranking is *affine per label*: $\rho_{\text{label}}(\boldsymbol{x}) = A_{label}\boldsymbol{x} + b_{label} \in \mathbb{N}^p$. Figure 2.(c) depicts the ranking found on the example. Ranks ordered with the lexicographic ordering $\ll$, the well-founded set is $(\mathbb{N}^p, \ll)$. This means that, by decreasing order, `start` comes first (2), then all the iterations of `loop` (1), and finally `stop` (0). The transitions involved to compute those constants are the transitions from `start` to `loop` and the transitions from `loop` to `stop`. Then, transitions from `loop` to `loop` (left, denoted τ_1 and right, denoted τ_2) are used to computed the second dimension of ρ_{loop}. *In the remainder, we will focus on the computation of the second dimension of ρ_{loop} $(x+y-2)$ from transitions τ_1 and τ_2.* We will write $\rho_{\text{loop}}(\boldsymbol{x})$ for $\rho_{\text{loop}}(\boldsymbol{x})[1]$ to simplify the presentation.

Positivity on reachable states The ranking must be positive on reachable states of `loop`. The set of $\boldsymbol{x}$ such that $\langle loop, \boldsymbol{x} \rangle$ is reachable from an initial state is called the *accessibility set* of `loop`. In general, we cannot compute it – this is the undecidable part of the analysis. Rather, we compute an over-approximation thanks to linear relation analysis [4, 9]. This set is called an *invariant* and will be denoted by $\mathcal{I}_{\text{loop}}$. Figure 2.(c) depicts the invariants on the program. All the challenge is to make the invariant close enough to the accessibility set so a ranking can be computed. In FKCC, the assertion $\boldsymbol{x} \models \mathcal{I}_{\text{loop}} \Rightarrow \rho_{\text{loop}}(\boldsymbol{x}) \geq 0$ translates to:

```
I_loop := [] -> {[x,y,x0,y0]: x>0 and y>0 and x <= x0 and y <= y0};
rank := positive_on I_loop;
```

Decreasing on transitions Now it remains to find a ranking decreasing on transitions τ_1 and τ_2. We first consider τ_1. The assertion $\boldsymbol{x} \models \mathcal{I}_{\text{loop}} \wedge x > y \Rightarrow \rho_{\text{loop}}(x-y, x, x_0, y_0) < \rho_{\text{loop}}(x, y, x_0, y_0)$ translates to:

```
tau1 := [] -> {[x,y,x0,y0]: x>y};
s1 := find rank s.t. rank - (rank . {[x,y,x0,y0]->[x-y,y,x0,y0]}) - 1
                         - positive_on (tau1*I_loop) = 0;
```

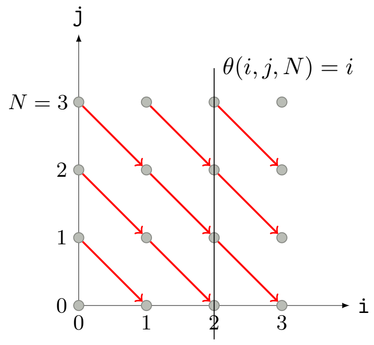
Similarly we compute a solution set `s2` from τ_2 and $\mathcal{I}_{\text{loop}}$. Finally, the ranking is found with the instruction `lexmin (s1*s2);`, which outputs the result:

```
[] -> {[1,1,0,0,-2]};
```

This corresponds to the dimension $x + y - 2$.

```
for $i$ := 0 to $N$
  for $j$ := 0 to $N$
    c[i+j] := c[i+j] + a[i]*b[j];
```

(a) Product of polynomials



(b) Iterations and schedule

Fig. 3. Scheduling example

4.2 Scheduling

Figure 3 depicts an example of program (a) computing the product of two polynomials specified by their array of coefficients `a` and `b`, and the iteration domain with the data dependence across iterations (b) and an example schedule θ prescribing a parallel execution by vertical waves, as discussed in Section 2.

Positivity Similarly to the ranking, the positivity condition (1) translates to:

```
iterations := [] -> { [i,j,N]: 0 <= i and i <= N  and 0 <= j and j <= N};
dependence := [] -> { [i,j,i',j',N]: 0 <= i and i <= N  and 0 <= j and
                       j <= N and 0 <= i' and i' <= N  and 0 <= j' and
                       j' <= N and i+j = i'+j' and i<i'};

# theta(i,j,N) >= 0 for any iteration (i,j,N)
theta := positive_on iterations;
```

Correctness We enhance the correctness condition (2) by making it possible to *select* the dependence to satisfy. For each dependence class d, we use a 0-1 variable ϵ_d. Here we have a single dependence class from S to S, so have only one 0-1 variable ϵ:

$$\theta(i',j',N) \geq \theta(i,j,N) + \epsilon \quad \forall (i,j,i',j') \in \Delta_N$$

On the ranking example, we would have four classes ($i = \mathit{start} \to \mathit{loop}, \tau_1, \tau_2, e = \mathit{loop} \to \mathit{stop}$). This makes it possible to choose which dependence class is satisfied ($\epsilon_d = 1$) or just respected ($\epsilon_d = 0$). This is the way multidimensional schedules are built [7]: on the termination example we would have $\epsilon_i = \epsilon_e = 1, \epsilon_{\tau_1} = \epsilon_{\tau_2} = 0$ for the first dimension, then $\epsilon_{\tau_1} = \epsilon_{\tau_2} = 1$ for the second dimension. Here it is kind of artificial, since we have a single dependence. However, the presentation generalizes easily to several dependence classes. This translates as:

```
parameters := {inv_eps,eps};

to_target := {[i,j,i',j',N]->[i',j',N]};
to_source := {[i,j,i',j',N]->[i,j,N]};

# s -> t ==> theta(s) <= theta(t) + eps, 0 <= eps <= 1
theta_correct := solve (theta . to_target) - (theta . to_source)
                       + {[i,j,i',j',N] -> -1*eps}
                       - (positive_on dependence) = 0;
theta_def := define theta with theta;
eps_correct := [] -> {[i]: 0 <= eps and eps <= 1 and inv_eps = 1-eps};
```

Here is the trick: parameters are forbidden to define Farkas terms; however parameters are perfectly allowed in summation. In that case, **the resolution interprets parameters as constants**. Hence the trick to set ϵ as a parameter and to put it in the summation by declaring an explicit affine form {`[i,j,i',j',N] -> -1*eps`}. We then keep the definition of theta coefficients in terms of Farkas multipliers (`theta_def`) and the domain of ϵ (`eps_correct`).

Optimality We seek a schedule θ with a minimal latency $\ell(\theta)$ (number of steps). When θ is an affine form, $\ell(\theta)$ may be bounded by an affine form $L(N)$ of the structure parameters [6]: $\ell(\theta) \leq L(N)$. This means that:

$$\forall (i,j) \in \mathcal{D}_N : \quad \theta(i,j,N) \leq L(N)$$

Which is, again, completely Farkas compliant. It remains to express $L(N)$, which have to be positive provided $\mathcal{D}_N$ is not empty i.e. $N \geq 0$. This translates to:

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

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

Finally, it remains to gather the constraints (positivity, correctness, optimality) to obtain 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);
```

By priority order, we want to (i) maximize the dependence satisfied (minimize `inv_eps`), then (ii) to minimize the latency (L(N) = latency_0*N + latency_1). This amounts to find the lexicographic minimum with variables ordered as `(inv_eps,latency_0,latency_1)`. Note that `eps` and `inv_eps` are parameters. Adding them to the variable list of `keep` has the effect to turn them to counters `eps_counter` and `inv_eps_counter`. We obtain the following result, pretty printed using the `-pretty` option:

```
theta_0 = 0
theta_1 = -1
theta_2 = 1
theta_3 = 0
latency_0 = 1
latency_1 = 0
eps_counter = 1
inv_eps_counter = 0
```

Hence $\theta(i, j, N) = N - j$, $L(N) = N$ and the dependence was satisfied (`eps_counter = 1`).

5 Conclusion

In this paper, we have presented FKCC, a scripting tool to prototype program analyses and transformations using the affine form of Farkas lemma. The script language of FKCC is powerful enough to write in a few lines tricky scheduling algorithms and termination analysis. The object representation (polyhedra, affine functions) is compatible with ISCC, a widespread polyhedral tool featuring manipulation of affine relations. FKCC provides features to generate ISCC code, and conversely, the output of ISCC might be injected in FKCC. This will allow to take profit of both worlds.

We believe that scripting tools are mandatory to evaluate rapidly research ideas. So far, Farkas lemma-based approaches were locked by two facts: (i) applying by hand Farkas Lemma is nearly impossible and (ii) implementing an analysis with Farkas lemma is usually tricky, time consuming and highly bug prone. With FKCC, computer scientists are now freed from these constraints.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: International Static Analysis Symposium (SAS'10) (2010)
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 101–113 (2008). https://doi.org/10.1145/1375581.1375595
3. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Springer-Verlag (ed.) CAV. pp. 420–432. No. 2725 in LNCS (2003)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM Symposium on Principles of Programming Languages (POPL'78). pp. 84–96. Tucson (Jan 1978)
5. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle **22**(3), 243–268 (1988)

6. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. International Journal of Parallel Programming **21**(5), 313–348 (Oct 1992). https://doi.org/10.1007/BF01407835
7. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: Multi-dimensional time. International Journal of Parallel Programming **21**(6), 389–420 (Dec 1992)
8. Feautrier, P., Lengauer, C.: Polyhedron model. In: Encyclopedia of Parallel Computing, pp. 1581–1592 (2011)
9. Gonnord, L.: Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires. Ph.D. thesis, Université Joseph Fourier - Grenoble (2007)
10. Verdoolaege, S.: Counting affine calculator and applications. In: IMPACT (2011)