# Farkas Lemma made easy

## Tool Demonstration

Christophe Alias
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Christophe.Alias@inria.fr

## 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 avalailable 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.

**ACM Reference Format:**
Christophe Alias. 2019. Farkas Lemma made easy: Tool Demonstration. In *Proceedings of 10th International Workshop on Polyhedral Compilation Techniques - in conjunction with HiPEAC 2020 (IMPACT 2020).* ACM, New York, NY, USA, 6 pages.

## 1 Introduction

Many program analyses require to handle universally quantified constraints such as $\forall x \in \mathcal{D} : \Phi(x)$, where $\mathcal{D}$ is a convex polyhedron and $\Phi$ is a conjunction of affine constraints. For instance, this occurs in loop scheduling [5, 6], loop tiling [3], program termination [2] or generation of loop invariants [4]. Farkas lemma – affine form – provides a way to get rid of that universal quantification, at the price of introducing quadratic terms. It is even possible to use Farkas lemma to turn universally quantified quadratic constraints into *existantially quantified affine constraints* [5, 6]. However, this requires tricky algebraic manipulations, notoriously difficult to experiment by hand and to implement.

In this tool demonstration, we present a scripting tool, FKCC [1] which makes it possible to manipulate easily Farkas lemma to benefit from those nice properties. Specifically, we will discuss the following points:

- A general formulation for the resolution of equations $\forall x : S(\vec{x}) = 0$ 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, is 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 tool demonstration 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. Finally, Section 4 concludes this paper and draws future research perspectives, then Annex A gives a real-life example of FKCC script to compute a Pluto-style tiling [3] for the Jacobi 1D kernel.

## 2 Farkas lemma in polyhedral compilation

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

**Lemma 2.1** (Farkas Lemma, affine form)**.** *Consider a nonempty convex polyhedron $\mathcal{P} = \{\vec{x},\ A\vec{x} + \vec{b} \geq 0\} \subseteq \mathbb{R}^n$ and an affine form $\phi : \mathbb{R}^n \longrightarrow \mathbb{R}$ such that $\phi(\vec{x}) \ \geq \ 0\ \forall \vec{x} \in \mathcal{P}$.*
*Then: $\exists \vec{\lambda} \geq \vec{0}, \lambda_0 \geq 0$ such that:*

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

Hence, Farkas lemma makes it possible to remove the quantification $\forall \vec{x} \in \mathcal{P}$ by encoding directly the positivity over $\mathcal{P}$ into the definition of $\phi$, thanks to the Farkas multipliers $\vec{\lambda}$ and $\lambda_0$. In the remainder, *Farkas terms* will be denoted by: $\mathfrak{F}(\lambda_0, \vec{\lambda}, A, \vec{b})(\vec{x}) = \ {}^t\vec{\lambda}(A\vec{x} + \vec{b}) + \lambda_0$. We now recall our theorem [1] to solve equations $\forall \vec{x} : S(\vec{x}) = 0$ where $S$ involves Farkas terms. The result is expressed as a conjunction of affine constraints, which is suited for integer linear programming:

**Theorem 2.2** (`solve`)**.** *Consider a summation $S(\vec{x}) = \vec{u} \cdot \vec{x} + v + \sum_i \mathfrak{F}(\lambda_{i0}, \vec{\lambda_i}, A_i, \vec{b_i})(\vec{x})$ of affine forms, including Farkas terms. Then:*

$$\forall \vec{x} :\ S(\vec{x}) = 0 \quad \textit{iff} \quad \left\{ \begin{array}{l} \vec{u} + \sum_i \ {}^tA_i\vec{\lambda}_i = \vec{0} \ \wedge \\ v + \sum_i \left(\vec{\lambda}_i \cdot \vec{b}_i + \lambda_{0i}\right) = 0 \end{array} \right.$$

***Application to scheduling*** Figure 1 depicts an example of a program (a) computing the product of two polynomials specified by their arrays of coefficients a and b, and the iteration domain with the data dependence across iterations (b) and an example schedule $\theta(i, j) = i$ prescribing a parallel execution by vertical waves. This paragraph reformulates the technique presented in [5] with our theorem 2.2. This formulation will directly inspire the FKCC syntax presented in the next section.

A schedule must be positive everywhere on its iteration domain:

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

Applying Farkas lemma, this translates to:

$$\exists \lambda_0 \geq 0, \vec{\lambda} \geq 0 \quad \text{s.t.} \quad \theta(i, j, N) = \mathfrak{F}(\lambda_0, \vec{\lambda}, A, \vec{b})(i, j, N) \tag{2}$$

Moreover, a schedule must *satisfy the data dependences* $(i, j) \to (i', j')$, abstracted by a dependence polyhedron $\Delta_N$:

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

This is equivalently written as the positivity of the affine form $(i, j, i', j', N) \mapsto \theta(i', j', N) - \theta(i, j, N) - 1$ over the convex polyhedron $\Delta_N$. Applying Farkas lemma:

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

Substituting $\theta$ using Equation (2), this translates to solving $\forall (i, j, i', j', N) : S(i, j, i', j', N) = 0$, where $S(i, j, i', j', N)$ is the summation:

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

Since $-\mathfrak{F}(\lambda_0, \vec{\lambda}, A, \vec{b}) = \mathfrak{F}(-\lambda_0, -\vec{\lambda}, A, \vec{b})$, we may apply theorem 2.2 to obtain a system of affine constraints with $\lambda_0, \vec{\lambda}, \mu_0, \vec{\mu}$. Linear programming may then be applied to find out the desired schedule [3, 6].

## 3 FKCC at a glance

This section outlines briefly the input syntax of FKCC on our motivating example. For a detailed description, the reader is referred to [1].

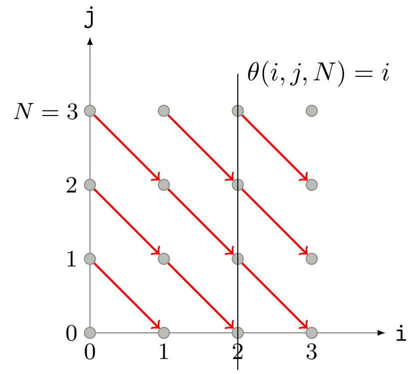
***Program, instructions, polyhedra*** An FKCC program consists of a sequence of instructions. There is no other control structure than the sequence. An instruction may assign an FKCC object (polyhedron, affine form or affine function) to an FKCC identifier, or may be an FKCC object alone. In the latter case, the FKCC object is streamed out to the standard output. FKCC objects are expressed with the same syntax as ISCC[7]:

```
parameters := {M,eps};
parametrized_iterations := [M] -> { [i]: 0 <= i and i <= M};
```

Parameters *must* be declared with the `parameters` construct. The parameters of a polyhedron *may* optionally be declared on preceding brackets `[M] -> ...`. The set intersection of two polyhedra `P` and `Q` is obtained with `P*Q`.

**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

**Figure 1.** Motivating example

***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 $\{\vec{x} \mid A\vec{x} + \vec{b} \geq 0\}$, then `theta` is defined as $\mathfrak{F}(\lambda_0, \vec{\lambda}, A, \vec{b})$ where $\lambda_0$ and $\vec{\lambda}$ are fresh positive variables. In this case, the polyhedron is *never* parametrized: the *parameters must be handled as variables.* Affine forms may be summed, scaled and composed with *affine functions*, typically to adjust the input dimension:

```
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'};
to_target := {[i,j,i',j',N] -> [i',j',N]};
to_source := {[i,j,i',j',N] -> [i,j,N]};
sum := (theta . to_target) - (theta . to_source) - 1
                          - positive_on dependence;
```

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`). The terms of the summation are simply separated with + and -, no parenthesis are allowed. Affine forms may also be stated explicitly:

```
sum_eps := (theta . to_target) - (theta . to_source)
         + {[i,j,i',j',N] -> -1*eps} - positive_on dependence;
```

***Resolution*** The main feature of FKCC is the resolution of equations $\forall \vec{x} : S(\vec{x}) = 0$ 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 2.2 (`solve`)),

```
[] -> {[lambda_0,lambda_1,lambda_2,lambda_3,lambda_4,lambda_5,
       lambda_6,lambda_7,lambda_8,lambda_9,lambda_10,lambda_11,
       lambda_12,lambda_13,lambda_14,lambda_15,lambda_16] :
#33 constraints! };
```

When the summation contains affine forms with parameters (as sum_eps), **the resolution interprets parameters as constants**. In particular, this makes it possible to tune dependence satisfaction: $\theta(i', j') \geq \theta(i, j) + \epsilon_d$ with $0 \leq \epsilon_d \leq 1$.

At this point, we need to recover the coefficients of our affine form theta in terms of $\vec{\lambda}$ (lambda_0, . . . , lambda_3) and $\lambda_0$ (lambda_4). Observe that theta($\vec{x}$) = $\mathfrak{F}(\lambda_0, \vec{\lambda}, A, \vec{b})(\vec{x})$, in turn equal to ${}^t\vec{\lambda}A\vec{x} + \vec{\lambda} \cdot \vec{b} + \lambda_0$. If the coefficients of theta are written: theta($\vec{x}$) = $\vec{\tau} \cdot \vec{x} + \tau_0$, we simply have: $\vec{\tau} = {}^t\vec{\lambda}A$ and $\tau_0 = \vec{\lambda} \cdot \vec{b} + \lambda_0$. 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] :
         lambda_0-lambda_1 = tau_0 and lambda_2-lambda_3 = tau_1 and
         lambda_1+lambda_3 = tau_2 and lambda_4 = tau_3};
```

The first coefficients tau_k define $\vec{\tau}$, the last one defines the constant $\tau_0$. 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 $\lambda$ to keep only $\vec{\tau}$ and $\tau_0$:

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

The result is a polyhedron with all the valid schedules:

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

All these steps may be applied at 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 chooses 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. As in ISCC, the lexicographic minimum of a polyhedron is obtained with the command lexmin. This example, however, does not admit a lexico-minimum solution. We can find a solution by forcing positive coefficients:

```
positive_theta := [] -> {[theta_0,theta_1,theta_2,theta_3]:
  theta_0 >= 0 and theta_1 >= 0 and theta_2 >= 0 and theta_3 >= 0};
lexmin (find theta s.t. sum = 0) * positive_theta;
```

Using the -pretty option, we obtain:

```
theta_0 = 1
theta_1 = 0
theta_2 = 0
theta_3 = 0
```

Which corresponds to the schedule $\theta(i, j, N) = i$. We point out that it is also possible to write in a few lines of FKCC the latency minimization as in [5]. A complete description may be found in [1].

# 4 Conclusion

FKCC is a scripting tool to prototype program analyses using the affine form of Farkas lemma. The script language of FKCC is powerful enough to write in a few lines sophisticated scheduling algorithms. 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: applying by hand Farkas Lemma is a pain; and implementing an analysis with Farkas lemma is notoriously time consuming and bug prone. With FKCC, computer scientists are now freed from these constraints.

Let the power of FKCC be with you!

# References


[1] Christophe Alias. 2019. fkcc: the Farkas Calculator. In *International Workshop on Tools for Automatic Program Analysis (TAPAS'19)*. Porto, Portugal.

[2] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *International Static Analysis Symposium (SAS'10)*.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. 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*. 101–113. https://doi.org/10.1145/1375581.1375595

[4] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV (LNCS)*, Springer-Verlag (Ed.). 420–432.

[5] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

[6] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[7] Sven Verdoolaege. 2011. Counting Affine Calculator and Applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*.

## A Pluto-style tiling with fkcc

We give here the FKCC script discussed in the demo session to compute a Pluto-style tiling [3] for the Jacobi-1D kernel. Note the sections expressing successively the data dependences ($\rightarrow$), the correctness condition ($\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \Rightarrow \phi_T(\vec{j}) - \phi_S(\vec{i}) \geq 0$), the laziness (minimal dependence distance $\vec{\delta}_{ST} = \phi_T(\vec{j}) - \phi_S(\vec{i})$), and finally the computation of non-null and linearly independent affine tiling hyperplanes in $\phi_S$ and $\phi_T$. Applying FKCC with the `-pretty` option, we obtain:

```
$ fkcc -pretty < pluto.fk
phi_S_0 = 1                     phi_S_0 = 2
phi_S_1 = 0                     phi_S_1 = 1
phi_S_2 = 0                     phi_S_2 = 0
phi_S_3 = 0                     phi_S_3 = 0
phi_S_4 = 0                     phi_S_4 = 0
phi_T_0 = 1                     phi_T_0 = 2
phi_T_1 = 0                     phi_T_1 = 1
phi_T_2 = 0                     phi_T_2 = 0
phi_T_3 = 0                     phi_T_3 = 0
phi_T_4 = 0                     phi_T_4 = 1
latency_0 = 0                   latency_0 = 0
latency_1 = 0                   latency_1 = 0
latency_2 = 1                   latency_2 = 2
```

Which corresponds to the affine tiling $\phi_S(t, i) = (t, 2t + i)$ and $\phi_T(t, i) = (t, 2t + i + 1)$ with a maximum dependence distance 1 for the first tiling hyperplane and 2 for the second tiling hyperplane.

```
#
# jacobi-1D non-perfect
#

# for (t = 1; t <= T; t++)
#   {
#     for (i = 1; i < N - 1; i++)
#       B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]); //S
#     for (i = 1; i < N - 1; i++)
#       A[i] = B [i];                                //T
#   }

D_S := [] -> { [t,i,T,N]: 1 <= t and t <= T and 1 <= i and i <= N-1};
D_T := [] -> { [t,i,T,N]: 1 <= t and t <= T and 1 <= i and i <= N-1};
phi_S := positive_on D_S;
phi_T := positive_on D_T;

#S --> T
Delta_ST := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T  and 1 <= i' and i' <= N-1 and
                                     t=t' and i=i'};

#S --> T (anti read(a[i-1]) --> write(a[i]))
Delta_ST_anti_1 := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T and 1 <= i' and i' <= N-1 and
                                            t=t' and i=i'-1};

#S --> T (anti read(a[i+1]) --> write(a[i]))
Delta_ST_anti_2 := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T and 1 <= i' and i' <= N-1 and
                                            t=t' and i=i'+1};

#T --> S, read a[i-1]
Delta_TS_1 := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T and 1 <= i' and i' <= N-1 and
                                       t+1=t' and i-1=i'};

#T --> S, read a[i]
Delta_TS_2 := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T and 1 <= i' and i' <= N-1 and
                                       t+1=t' and i=i'};

#T --> S, read a[i+1]
Delta_TS_3 := [] -> { [t,i,t',i',T,N]: 1 <= t and t <= T  and 1 <= i and i <= N-1 and 1 <= t' and t' <= T and 1 <= i' and i' <= N-1 and
                                       t+1=t' and i+1=i'};
```

```
#
# Correctness: s -> t ==> phi(s) <= phi(t)
#

to_target := {[t,i,t',i',T,N]->[t',i',T,N]};
to_source := {[t,i,t',i',T,N]->[t,i,T,N]};

phi_correct :=
  # S --> T
  (find phi_S,phi_T s.t. (phi_T . to_target) - (phi_S . to_source) - positive_on Delta_ST = 0) *
  # S --> T, (anti read(a[i-1]) --> write(a[i]))
  (find phi_S,phi_T s.t. (phi_T . to_target) - (phi_S . to_source) - positive_on Delta_ST_anti_1 = 0) *
  # S --> T, (anti read(a[i+1]) --> write(a[i]))
  (find phi_S,phi_T s.t. (phi_T . to_target) - (phi_S . to_source) - positive_on Delta_ST_anti_2 = 0) *
  # T --> S, read a[i-1]
  (find phi_S,phi_T s.t. (phi_S . to_target) - (phi_T . to_source) - positive_on Delta_TS_1 = 0) *
  # T --> S, read a[i]
  (find phi_S,phi_T s.t. (phi_S . to_target) - (phi_T . to_source) - positive_on Delta_TS_2 = 0) *
  # T --> S, read a[i+1]
  (find phi_S,phi_T s.t. (phi_S . to_target) - (phi_T . to_source) - positive_on Delta_TS_3 = 0);

#
# Efficiency: s -> t ==> phi(t) - phi(s) <= latency(N), then min latency(N)
#

# L(N) >= 0 on the parameter domain
latency := positive_on ([] -> {[T,N]: T >= 0 and N >= 0});
latency_def := define latency with latency;
to_param := {[t,i,t',i',T,N] -> [T,N]};

# phi(t) - phi(s) <= L(N) \forall s --> t
phi_bounded :=
  # S --> T
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_T . to_target) + (phi_S . to_source) - positive_on Delta_ST = 0) *
  # S --> T, (anti read(a[i-1]) --> write(a[i]))
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_T . to_target) + (phi_S . to_source) - positive_on Delta_ST_anti_1 = 0) *
  # S --> T, (anti read(a[i+1]) --> write(a[i]))
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_T . to_target) + (phi_S . to_source) - positive_on Delta_ST_anti_2 = 0) *
  # T --> S, read a[i-1]
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_S . to_target) + (phi_T . to_source) - positive_on Delta_TS_1 = 0) *
  # T --> S, read a[i]
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_S . to_target) + (phi_T . to_source) - positive_on Delta_TS_2 = 0) *
  # T --> S, read a[i+1]
  (find latency,phi_S,phi_T s.t. latency . to_param - (phi_S . to_target) + (phi_T . to_source) - positive_on Delta_TS_3 = 0);

#
# First hyperplane: avoid the null solution
#

phi_filter_level_1 := [] -> {[phi_S_0,phi_S_1,phi_S_2,phi_S_3,phi_S_4,phi_T_0,phi_T_1,phi_T_2,phi_T_3,phi_T_4]:

#phi_S: positive coefficients + no parameters
phi_S_0 >= 0 and
phi_S_1 >= 0 and
phi_S_2 = 0 and
phi_S_3 = 0 and
phi_S_4 >= 0 and

#phi_T: positive coefficients + no parameters
phi_T_0 >= 0 and
phi_T_1 >= 0 and
phi_T_2 = 0 and
phi_T_3 = 0 and
phi_T_4 >= 0 and
```

```
#phi_S != 0 and phi_T != 0
phi_S_0 + phi_S_1 + phi_S_2 + phi_S_3 >= 1 and
phi_S_0 + phi_T_1 + phi_T_2 + phi_T_3 >= 1
};

all_level_1 := keep latency_0,latency_1,latency_2,phi_S_0,phi_S_1,phi_S_2,phi_S_3,phi_S_4,phi_T_0,phi_T_1,phi_T_2,phi_T_3,phi_T_4
              in phi_correct * phi_bounded * phi_filter_level_1;

lexmin all_level_1;

#
# Second hyperplane: avoid the null solution + linear independence with the first hyperplane
#

phi_filter_level_2 := [] -> {[phi_S_0,phi_S_1,phi_S_2,phi_S_3,phi_S_4,phi_T_0,phi_T_1,phi_T_2,phi_T_3,phi_T_4]:

#phi_S: positive coefficients + no parameters + lin. ind (coef(i) > 0)
phi_S_0 >= 0 and
phi_S_1 > 0 and
phi_S_2 = 0 and
phi_S_3 = 0 and
phi_S_4 >= 0 and

#phi_T: positive coefficients + no parameters + lin. ind (coef(i) > 0)
phi_T_0 >= 0 and
phi_T_1 > 0 and
phi_T_2 = 0 and
phi_T_3 = 0 and
phi_T_4 >= 0 and

#phi_S != 0 and phi_T != 0
phi_S_0 + phi_S_1 + phi_S_2 + phi_S_3 >= 1 and
phi_S_0 + phi_T_1 + phi_T_2 + phi_T_3 >= 1
};

all_level_2 := keep latency_0,latency_1,latency_2,phi_S_0,phi_S_1,phi_S_2,phi_S_3,phi_S_4,phi_T_0,phi_T_1,phi_T_2,phi_T_3,phi_T_4
              in phi_correct * phi_bounded * phi_filter_level_2;

lexmin all_level_2;
```