Farkas Lemma made easy
Christophe Alias

To cite this version:
Christophe Alias. Farkas Lemma made easy. 10th International Workshop on Polyhedral Compilation Techniques (IMPACT 2020), Jan 2020, Bologna, Italy. pp.1-6. hal-02422033

HAL Id: hal-02422033
https://inria.hal.science/hal-02422033
Submitted on 8 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 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.

ACM Reference Format:

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 existentially 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 \( \forall x : S(\overline{x}) = 0 \) 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 \( \forall \overline{x} : S(\overline{x}) = 0 \) 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 non-empty convex polyhedron \( \mathcal{P} = (\overline{A}, A\overline{x} + \overline{b} \geq 0) \subseteq \mathbb{R}^n \) and an affine form \( \phi : \mathbb{R}^n \rightarrow \mathbb{R} \) such that \( \phi(\overline{x}) \geq 0 \forall \overline{x} \in \mathcal{P} \).

Then: \( \exists \overline{\lambda} \geq 0, \lambda_0 \geq 0 \) such that:

\[
\phi(\overline{x}) = \overline{\lambda}(A\overline{x} + \overline{b}) + \lambda_0 \quad \forall \overline{x}
\]

Hence, Farkas lemma makes it possible to remove the quantification \( \forall \overline{x} \in \mathcal{P} \) by encoding directly the positivity over \( \mathcal{P} \) into the definition of \( \phi \), thanks to the Farkas multipliers \( \overline{\lambda} \) and \( \lambda_0 \). In the remainder, Farkas terms will be denoted by: \( \overline{\gamma}(\lambda_0, \overline{\lambda}, A, \overline{b})(\overline{x}) = \overline{\lambda}(A\overline{x} + \overline{b}) + \lambda_0 \). We now recall our theorem [1] to solve equations \( \forall \overline{x} : S(\overline{x}) = 0 \) 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 \( S(\overline{x}) = \overline{u} \cdot \overline{x} + v + \sum_i \overline{c}(\lambda_{i0}, \overline{\lambda}_i, A_i, \overline{b}_i)(\overline{x}) \) of affine forms, including Farkas terms. Then:

\[
\forall \overline{x} : S(\overline{x}) = 0 \quad \text{iff} \quad \begin{cases}
\overline{u} + \sum_i t_i \overline{A}_i \overline{\lambda}_i = \overline{0} & \land \\
v + \sum_i \left( \overline{\lambda}_i \cdot \overline{b}_i + \lambda_{i0} \right) = 0
\end{cases}
\]
Applying Farkas lemma, this translates to:

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

This is equivalently written as the positivity of the affine convex polyhedron:

$$\forall (i, j) \in \mathcal{D}_N$$

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

$$\theta(i', j', N) > \theta(i, j, N) \quad \forall (i, j, i', j') \in \Delta_N$$  \hspace{1cm} (3)

An affine form may be defined as a Farkas term. This is obtained with the input dimension:

$$\text{solve } \sum = 0;$$

$$\text{for } i := 0 \text{ to } N$$

$$\text{for } j := 0 \text{ to } N$$

$$c[i+j] := c[i+j] + a[i]b[j];$$

(a) Product of polynomials

Figure 1. Motivating example

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

$$\text{theta := positive_on iterations; }$$

If iterations is \( \{ x | Ax + b \geq 0 \} \), then theta is defined as \( \theta(\lambda_0, \lambda, A, b) \) where \( \lambda_0 \) and \( \lambda \) 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:

$$\text{dependence := \{ [i, j, i', j', N] : 0 <= i \text{ and } i <= N \text{ and } 0 <= j \text{ and } j <= N \text{ and } 0 <= i' \text{ and } i' <= N \text{ and } 0 <= j' \text{ and } j' <= N \text{ and } i+j = i'+j \text{ and } i < i' \}; }$$

$$\text{to_target := \{ [i, j, i', j', N] \} \rightarrow [i, j, i', j', N]; }$$

$$\text{to_source := \{ [i, j, i', j', N] \} \rightarrow [i, j, N]; }$$

$$\text{sum := (theta . to_target) - (theta . to_source) - 1 - positive_on dependence; }$$

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 \( \{ [i, j, i', j', N] \} \rightarrow -1 \). The terms of the summation are simply separated with + and –, no parenthesis are allowed. Affine forms may also be stated explicitly:

$$\text{sum_eps := (theta . to_target) - (theta . to_source); }$$

$$\text{solve sum = 0; }$$

The result is a polyhedron with Farkas multipliers (obtained after applying Theorem 2.2 (solve)):
When the summation contains affine forms with parameters (as \texttt{sum_eps}), the resolution interprets parameters as constants. In particular, this makes it possible to tune dependence satisfaction: \( \theta'(i',j') \geq \theta(i,j) + \varepsilon_d \) with \( 0 \leq \varepsilon_d \leq 1 \).

At this point, we need to recover the coefficients of our affine form \( \theta \) in terms of \( \lambda (\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4) \) and \( \lambda_0 (\lambda_0, \ldots, \lambda_4) \). Observe that \( \theta(x) = \theta'_0 \lambda_A x + \theta'_1 \lambda_B x + \theta'_2 \lambda_C x + \theta'_3 \lambda_D x \), in turn equal to \( \theta'_0 \lambda_A x + \theta'_1 \lambda_B x + \theta'_2 \lambda_C x + \theta'_3 \lambda_D x \). If the coefficients of \( \theta \) are written: \( \theta(x) = \tau \cdot x + r_0 \), we simply have: \( \tau = \tau'_0 \lambda_A \) and \( r_0 = \lambda_A \cdot \lambda_B + \lambda_0 \). This is obtained with define:

define theta with tau;

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

\[
[] \rightarrow \{ \lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \tau_0, \tau_1, \tau_2, \tau_3, \tau_4 \}.
\]

The first coefficients \( \tau_k \) define \( \tau \), the last one defines the constant \( r_0 \). On our example, \( \tau(i,j,N) = \tau_0 \cdot i + \tau_1 \cdot j + \tau_2 \cdot N + \tau_3 \). Now we may gather the results and eliminate the \( \lambda \) to keep only \( \tau \) and \( r_0 \):

\[
\begin{align*}
\text{keep } \tau_0, \tau_1, \tau_2, \tau_3, \tau_4 \text{ in } \\
&((\text{solve sum = 0})*\text{define theta with tau});
\end{align*}
\]

The result is a polyhedron with all the valid schedules:

\[
[] \rightarrow \{ \tau_0, \tau_1, \tau_2, \tau_3, \tau_4 \}:
\]

\[
\begin{align*}
\tau_3 & \gg 0 \text{ and } (\tau_0 \cdot \tau_1) \cdot \tau_2 \gg 0 \text{ and } \\
(\neg \tau_0) \cdot (\neg \tau_1) & \gg 0 \text{ and } \\
\tau_1 \cdot \tau_2 & \gg 0 \text{ and } \tau_2 \cdot \tau_4 \gg 0;
\end{align*}
\]

All these steps may be applied at once with the \texttt{find} command:

\[
\begin{align*}
\text{find } \theta \text{ s.t. sum = } 0; \\
\text{The coefficients are automatically named } \theta_0, \theta_1, \theta_2, \theta_3, \text{ etc with the same convention as define. We point out that define choose fresh names for coefficients (e.g. } \tau_4, \tau_5 \text{ on the second time with ‘’}\tau\text{’’) whereas find always chooses the same names. Hence find would be preferred when deriving separately constraints on the same coefficients of } \theta. \text{ find may filter the coefficients for several affine forms expressed as Farkas terms in a summation:}
\end{align*}
\]

\[
\begin{align*}
\text{find } \theta_S, \theta_T \text{ s.t. } \\
\theta_T . \text{to_target} - \theta_S . \text{to_source} - 1 \rightarrow \text{positive_on dependences_from_S_to_T} = 0;
\end{align*}
\]

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 \texttt{lexmin}. This example, however, does not admit a lexico-minimum solution. We can find a solution by forcing positive coefficients:

\[
\begin{align*}
\text{positive_theta := } [] \rightarrow \{ \theta_0, \theta_1, \theta_2, \theta_3 \}:
\theta_0 \gg 0 \text{ and } \theta_1 \gg 0 \text{ and } \theta_2 \gg 0 \text{ and } \theta_3 \gg 0; \\
\text{lexmin (find theta s.t. sum = 0) * positive_theta;}
\end{align*}
\]

Using the \texttt{-pretty} option, we obtain:

\[
\begin{align*}
\theta_0 = 1 \\
\theta_1 = 0 \\
\theta_2 = 0 \\
\theta_3 = 0
\end{align*}
\]

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 \texttt{fkcc} the latency minimization as in [5]. A complete description may be found in [1].

## 4 Conclusion

\texttt{fkcc} is a scripting tool to prototype program analyses using the affine form of Farkas lemma. The script language of \texttt{fkcc} is powerful enough to write in a few lines sophisticated scheduling algorithms. The object representation (polyhedral, affine functions) is compatible with \texttt{iscc}, a widespread polyhedral tool featuring manipulation of affine relations. \texttt{fkcc} provides features to generate \texttt{iscc} code, and conversely, the output of \texttt{iscc} might be injected in \texttt{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 \texttt{fkcc}, computer scientists are now freed from these constraints.

Let the power of \texttt{fkcc} be with you!

## References


A Pluto-style tiling with \texttt{fkcc}

We give here the \texttt{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 (\(\langle S, i \rangle \rightarrow \langle T, j \rangle \Rightarrow \phi_T(j) - \phi_S(i) \geq 0\)), the laziness (minimal dependence distance \(\delta_{ST} = \phi_T(j) - \phi_S(i)\)), and finally the computation of non-null and linearly independent affine tiling hyperplanes in \(\phi_S\) and \(\phi_T\). Applying \texttt{fkcc} with the \texttt{-pretty} option, we obtain:

\[
\begin{align*}
\text{phi}_S & : \{0, 1, 2, 3, 4\} \\
\text{phi}_T & : \{0, 1, 2\} \\
\text{latency}_0 & : 0 \\
\text{latency}_1 & : 1 \\
\text{latency}_2 & : 2
\end{align*}
\]

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.

```bash
$ 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
latency_0 = 0
latency_1 = 0
latency_2 = 1
```

# # jacobi-1D non-perfect
# #
# # for (t = 1; t <= T; t++)
# # {
# #   for (i = 1; i < N - 1; i++)
# #     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;

D_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'\};
D_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\};
D_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\};

D_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=t'+1 and i=i'\};
D_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=t'+1 and i=i'\};

D_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=t'+1 and i=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 ([I] -> {[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 := {[I] -> {[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_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;