Galois Connections for Flow Algebras

Piotr Filipiuk, Michał Terepeta, Hanne Riis Nielson, and Flemming Nielson

Technical University of Denmark
{pifi,mtte,riis,nielson}@imm.dtu.dk

Abstract. We generalise *Galois connections* from complete lattices to flow algebras. *Flow algebras* are algebraic structures that are less restrictive than idempotent semirings in that they replace distributivity with monotonicity and dispense with the annihilation property; therefore they are closer to the approach taken by Monotone Frameworks and other classical analyses. We present a generic framework for static analysis based on flow algebras and program graphs. Program graphs are often used in Model Checking to model concurrent and distributed systems. The framework allows to *induce* new flow algebras using Galois connections such that correctness of the analyses is preserved. The approach is illustrated for a *mutual exclusion* algorithm.

1 Introduction

In the classical approach to static analysis we usually use the notion of Monotone Frameworks [9, 1] that work over flow graphs as an abstract representation of a program. A Monotone Framework, primarily used for data-flow analysis [10], consists of a complete lattice describing the properties of the system (without infinite ascending chains) and transfer functions over that lattice (that are monotone). When working with complete lattices, one can take advantage of Galois connections to induce new analysis or over-approximate them [6]. Recall that a Galois connection is a correspondence between two complete lattices that consists of an abstraction and concretisation functions. It is often used to move an analysis from a computationally expensive lattice to a less costly one and plays a crucial role in abstract interpretation [5].

In this paper we introduce a similar framework that uses flow algebras to define analyses. Flow algebras are algebraic structures consisting of two monoids [15] quite similar to idempotent semirings [7], which have already been used in software analysis [3, 14]. However, flow algebras are less restrictive and allow to directly express some of the classical analysis, which is simply not possible with idempotent semirings. Furthermore as representation of the system under consideration we use *program graphs*, in which actions label the edges rather than the nodes. The main benefit of using program graphs is that we can model concurrent systems in a straightforward manner. Moreover since a model of a concurrent system is also a program graph, all the results are applicable both in the sequential as well as in the concurrent setting.

We also define both the Meet Over all Paths (*MOP*) solution of the analysis as well as a set of constraints that can be used to obtain the Maximal Fixed Point (*MFP*) solution. By establishing that the solutions of the constraints constitute a Moore family, we know that there always is a least (i.e. best) solution. Intuitively the main difference between *MOP* and *MFP* is that the former expresses what we would like to compute, whereas the latter is sometimes less accurate but computable in some cases where *MOP* is not. Finally we establish that they coincide in case of distributive analyses.

We also extend the notion of Galois connections to flow algebras and program graphs. This allows us to easily create new analyses based on existing ones. In particular we can create Galois connections between the collecting semantics (defined in terms of our framework) and various analyses, which ensures their semantic correctness.

Finally we apply our results to a variant of the Bakery mutual exclusion algorithm [11]. By inducing an analysis from the collecting semantics and a Galois insertion, we are able to prove the correctness of the algorithm. Thanks to our previous developments we know that the analysis is semantically correct.

The structure of the paper is as follows. In Section 2 we introduce the flow algebras and then we show how Galois connections are defined for them in Section 3. We perform a similar development for program graphs by defining them in Section 4, presenting how to express analysis using them in Section 5 and then describing Galois connections for program graphs in Section 6. Finally we present a motivating example of our approach in Section 7 and conclude in Section 8.

2 Flow Algebra

In this section we introduce the notion of a flow algebra. As already mentioned, it is an algebraic structure that is less restrictive than an idempotent semiring. Recall that an idempotent semiring is a structure of the form $(S, \oplus, \otimes, 0, 1)$, such that $(S, \oplus, 0)$ is an idempotent commutative monoid, $(S, \otimes, 1)$ is a monoid, where $\otimes$ distributes over $\oplus$ and 0 is an annihilator with respect to multiplication. In contrast to idempotent semirings flow algebras do not require the distributivity and annihilation properties. Instead we replace the first one with a monotonicity requirement and dispense with the second one. A flow algebra is formally defined as follows.

Definition 1. *A flow algebra is a structure of the form* $(F, \oplus, \otimes, 0, 1)$ *such that:*

- $(F, \oplus, 0)$ *is an idempotent commutative monoid:*
  - $(f_1 \oplus f_2) \oplus f_3 = f_1 \oplus (f_2 \oplus f_3)$
  - $0 \oplus f = f \oplus 0 = f$
  - $f_1 \oplus f_2 = f_2 \oplus f_1$
  - $f \oplus f = f$
- $(F, \otimes, 1)$ *is a monoid:*
  - $(f_1 \otimes f_2) \otimes f_3 = f_1 \otimes (f_2 \otimes f_3)$
  - $1 \otimes f = f \otimes 1 = f$

– $\otimes$ *is monotonic in both arguments:*
  - $f_1 \leq f_2 \;\Rightarrow\; f_1 \otimes f \leq f_2 \otimes f$
  - $f_1 \leq f_2 \;\Rightarrow\; f \otimes f_1 \leq f \otimes f_2$

where $f_1 \leq f_2$ *if and only if* $f_1 \oplus f_2 = f_2$.

In a flow algebra all finite subsets $\{f_1, \cdots, f_n\}$ have a least upper bound; it is given by $0 \oplus f_1 \oplus \cdots \oplus f_n$.

Definition 2. *A distributive flow algebra is a flow algebra* $(F, \oplus, \otimes, 0, 1)$*, where* $\otimes$ *distributes over* $\oplus$ *on both sides, i.e.*

$$f_1 \otimes (f_2 \oplus f_3) = (f_1 \otimes f_2) \oplus (f_1 \otimes f_3)$$
$$(f_1 \oplus f_2) \otimes f_3 = (f_1 \otimes f_3) \oplus (f_2 \otimes f_3)$$

We also say that a flow algebra is strict if $0 \otimes f = 0 = f \otimes 0$.

Fact 1 *Every idempotent semiring is a strict and distributive flow algebra.*

We consider flow algebras because they are closer to Monotone Frameworks, and other classical static analyses. Restricting our attention to semirings rather than flow algebras would mean restricting attention to strict and distributive frameworks. Note that the classical bit-vector frameworks [12] are distributive, but not strict; hence they are not directly expressible using idempotent semirings.

Definition 3. *A complete flow algebra is a flow algebra* $(F, \oplus, \otimes, 0, 1)$*, where* F *is a complete lattice; we write* $\bigoplus$ *for the least upper bound. It is affine [12] if for all non-empty subsets* $F' \neq \emptyset$ *of* F

$$f \otimes \bigoplus F' = \bigoplus \{f \otimes f' \mid f' \in F'\}$$

$$\bigoplus F' \otimes f = \bigoplus \{f' \otimes f \mid f' \in F'\}$$

Furthermore, it is completely distributive if it is affine and strict.

If the complete flow algebra satisfies the Ascending Chain Condition [12] then it is affine if and only if it is distributive. The proof is analogous to the one presented in Appendix A of [12].

Example 1. As an example let us consider a complete lattice $L \to L$ of monotone functions over the complete lattice L. Then we can easily define a flow algebra for forward analyses, by taking $(L \to L, \sqcup, \fatsemi, \lambda f.\bot, \lambda f.f)$ where $(f_1 \fatsemi f_2)(l) = f_2(f_1(l))$ for all $l \in L$. It is easy to see that all the laws of a complete flow algebra are satisfied. If we restrict the functions in $L \to L$ to be distributive, we obtain a distributive and complete flow algebra. Note that it can be used to define forward data-flow analyses such as reaching definitions [1, 12].

3 Galois Connections for Flow Algebras

Let us recall that *Galois connection* is a tuple (L, α, γ, M) such that L and M are complete lattices and α, γ are monotone functions (called abstraction and concretisation functions) that satisfy $\alpha \circ \gamma \sqsubseteq \lambda m.m$ and $\gamma \circ \alpha \sqsupseteq \lambda l.l$. A *Galois insertion* is a Galois connection such that $\alpha \circ \gamma = \lambda m.m$. In this section we will present them in the setting of flow algebras.

In order to extend the Galois connections for flow algebras, we need to define what it means for a flow algebra to be an upper-approximation of another flow algebra. In other words we need to impose certain conditions on $\otimes$ operator and 1 element of the less precise flow algebra. The requirements are presented in the following definition.

Definition 4. *For a Galois connection (L, α, γ, M) we say that the flow algebra $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ is an upper-approximation of $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ if*

$$\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) \sqsubseteq_M m_1 \otimes_M m_2$$
$$\alpha(1_L) \sqsubseteq_M 1_M$$

If we have equalities in the above definition, then we say that the flow algebra $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ is *induced* from $(L, \oplus_L, \otimes_L, 0_L, 1_L)$.

Example 2. Assume that we have a Galois connection (L, α, γ, M) between complete lattices L and M. We can easily construct $(L \to L, \alpha', \gamma', M \to M)$ which is a Galois connection between monotone function spaces on those lattices (for more details about this construction please consult Section 4.4 of [12]), where α', γ' are defined as

$$\alpha'(f) = \alpha \circ f \circ \gamma$$
$$\gamma'(g) = \gamma \circ g \circ \alpha$$

When both $(M \to M, \oplus_M, \otimes_M, 0_M, 1_M)$ and $(L \to L, \oplus_L, \otimes_L, 0_L, 1_L)$ are forward analyses as in Example 1, we have

$$\begin{aligned}
\alpha'(\gamma'(g_1) \otimes_L \gamma'(g_2)) &= \alpha'((\gamma \circ g_1 \circ \alpha) \fatsemi (\gamma \circ g_2 \circ \alpha)) \\
&= \alpha'(\gamma \circ g_2 \circ \alpha \circ \gamma \circ g_1 \circ \alpha) \\
&\sqsubseteq \alpha'(\gamma \circ g_2 \circ g_1 \circ \alpha) \\
&= \alpha \circ \gamma \circ g_2 \circ g_1 \circ \alpha \circ \gamma \\
&\sqsubseteq g_2 \circ g_1 \\
&= g_1 \otimes_M g_2
\end{aligned}$$

$$\alpha'(1_L) = \alpha \circ \lambda l.l \circ \gamma = \alpha \circ \gamma \sqsubseteq \lambda m.m = 1_M$$

Hence a flow algebra over $M \to M$ is a upper-approximation of the flow algebra over $L \to L$. Note that in case of a Galois insertion the flow algebra over $M \to M$ is induced.

Definition 4 requires a bit of care. Given a flow algebra $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ and a Galois connection (L, α, γ, M) it is tempting to define $\otimes_M$ by $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ and 1_M by $1_M = \alpha(1_L)$. However, it is not generally the case that $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ will be a flow algebra. This motivates the following development.

Lemma 1. *Let $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ be a flow algebra, (L, α, γ, M) be a Galois insertion, define $\otimes_M$ by $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ and 1_M by $1_M = \alpha(1_L)$. If*

$$1_L \in \gamma(M) \quad \textit{and} \quad \otimes_L : \gamma(M) \times \gamma(M) \to \gamma(M)$$

for all m_1, m_2 then $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ is a flow algebra (where $\oplus_M$ is $\sqcup_M$ and 0_M is $\bot_M$).

Proof. We need to ensure that $\otimes_M$ is associative

$$\begin{aligned}
(m_1 \otimes_M m_2) \otimes_M m_3 &= \alpha(\gamma(\alpha(\gamma(m_1) \otimes_L \gamma(m_2))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3)) \\
m_1 \otimes_M (m_2 \otimes_M m_3) &= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m_2) \otimes_L \gamma(m_3)))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m')))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3))
\end{aligned}$$

and similarly we need to show that 1_M is a neutral element for $\otimes_M$

$$\begin{aligned}
1_M \otimes m &= \alpha(1_L) \otimes_M m \\
&= \alpha(\gamma(\alpha(1_L)) \otimes_L \gamma(m)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m)) \\
&= \alpha((\gamma(m') \otimes_L \gamma(m)) \\
&= \alpha((1_L \otimes_L \gamma(m)) \\
&= \alpha(\gamma(m)) \\
&= m
\end{aligned}$$

where $1_L = \gamma(m')$ for some m'. The remaining properties of flow algebra hold trivially. □

The above requirements can be expressed in a slightly different way. This is presented by the following two lemmas.

Lemma 2. *For flow algebras $(L, \oplus_L, \otimes_L, 0_L, 1_L)$, $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:*

1. $1_L = \gamma(1_M)$
2. $\alpha(1_L) = 1_M$ *and* $1_L \in \gamma(M)$

Lemma 3. *For flow algebras $(L, \oplus_L, \otimes_L, 0_L, 1_L)$, $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:*

1. $\forall m_1, m_2 : \gamma(m_1) \otimes_L \gamma(m_2) = \gamma(m_1 \otimes_M m_2)$
2. $\forall m_1, m_2 : \alpha(\gamma(m_1) \otimes_L \gamma(m_2)) = m_1 \otimes_M m_2$ *and* $\otimes_L : \gamma(M) \times \gamma(M) \to \gamma(M)$

4 Program Graphs

This section introduces program graphs, a representation of software (hardware) systems that is often used in model checking [2] to model concurrent and distributed systems. Compared to the classical flow graphs [10, 12], the main difference is that in the program graphs the actions label the edges rather than the nodes.

Definition 5. *A program graph over a space S has the form*

$$(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$$

where

- Q *is a finite set of states;*
- Σ *is a finite set of actions;*
- $\rightarrow \subseteq \mathsf{Q} \times \Sigma \times \mathsf{Q}$ *is a transition relation;*
- $\mathsf{Q}_I \subseteq \mathsf{Q}$ *is a set of initial states;*
- $\mathsf{Q}_F \subseteq \mathsf{Q}$ *is a set of final states; and*
- $\mathcal{A} : \Sigma \rightarrow S$ *specifies the meaning of the actions.*

A concrete program graph is a program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$, where **Dom** *is the set of all configurations of a program, and $\mathcal{A} = \mathcal{T}$ where $\mathcal{T}$ is the semantic function. An abstract program graph is a program graph where S is a complete flow algebra.*

Now we can define the collecting semantics [5, 12] of a concrete program graph in terms of a flow algebra. This can be used to establish the semantic correctness of an analysis by defining a Galois connection between the collecting semantics and the analysis.

Definition 6. *We define the collecting semantics of a program graph using the flow algebra $(\mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom}), \cup, \fatsemi, \lambda.\emptyset, \lambda d.d)$, by*

$$\mathcal{A}[\![a]\!](S) = \{\mathcal{T}[\![a]\!](s) \mid s \in S \wedge \mathcal{T}[\![a]\!](s) \textit{ is defined}\}$$

where **Dom** *is the set of all configurations of a program and $\mathcal{T}$ is the semantic function.*

Now let us consider a number of processes each specified as a program graph $PG_i = (\mathsf{Q}_i, \Sigma_i, \rightarrow_i, \mathsf{Q}_{Ii}, \mathsf{Q}_{Fi}, \mathcal{A}_i, S)$ that are executed independently of one another except that they can exchange information via shared variables. The combined program graph $PG = PG_1 \mid\mid\mid \cdots \mid\mid\mid PG_n$ expresses the interleaving between n processes.

Definition 7. *The interleaved program graph over S*

$$\mathsf{PG} = \mathsf{PG}_1 \mid\mid\mid \cdots \mid\mid\mid \mathsf{PG}_n$$

is defined by $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$ where

– $\mathsf{Q} = \mathsf{Q}_1 \times \cdots \times \mathsf{Q}_n$,
– $\Sigma = \Sigma_1 \uplus \cdots \uplus \Sigma_n$ *(disjoint union)*,
– $\langle q_1, \cdots, q_i, \cdots, q_n \rangle \xrightarrow{a} \langle q_1, \cdots, q_i', \cdots, q_n \rangle$ *if* $q_i \xrightarrow{a}_i q_i'$,
– $\mathsf{Q}_I = \mathsf{Q}_{I1} \times \cdots \times \mathsf{Q}_{In}$,
– $\mathsf{Q}_F = \mathsf{Q}_{F1} \times \cdots \times \mathsf{Q}_{Fn}$, *and*
– $\mathcal{A}[\![a]\!] = \mathcal{A}_i[\![a]\!]$ *if* $a \in \Sigma_i$.

Note that $\mathcal{A}_i : \Sigma_i \to S$ *for all* i *and hence* $\mathcal{A} : \Sigma \to S$.

Analogously to the previous definition, we say that a concrete interleaved program graph is an interleaved program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$, *and* $\mathcal{A} = \mathcal{T}$ *where* $\mathcal{T}$ *is the semantic function. An abstract interleaved program graph is an interleaved program graph where* S *is a complete flow algebra.*

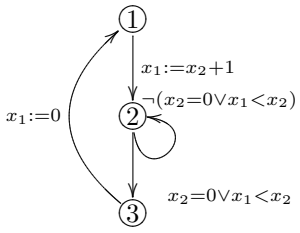
The application of this definition is presented in the example below, where we model the Bakery mutual exclusion algorithm. Note that the ability to create interleaved program graphs allows us to model concurrent systems using the same methods as in the case of sequential programs. This will be used to analyse and verify the algorithm in Section 7.

Example 3. As an example we consider a variant of the Bakery algorithm for two processes. Let P_1 and P_2 be the two processes, and x_1 and x_2 be two shared variables both initialised to 0. The algorithm is as follows

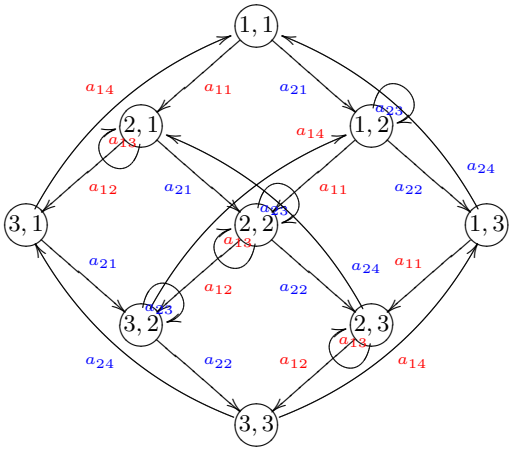
```
do true ->                                  ||  do true ->
    x1 := x2 + 1;                           ||      x2 := x1 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->           ||      do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip                                ||          skip
    od;                                     ||      od;
    critical section                        ||      critical section
    x1 := 0                                 ||      x2 := 0
od                                          ||  od
```

The variables x_1 and x_2 are used to resolve the conflict when both processes want to enter the critical section. When x_i is equal to zero, the process P_i is not in the critical section and does not attempt to enter it — the other one can safely proceed to the critical section. Otherwise, if both shared variables are non-zero, the process with smaller "ticket" (i.e. value of the corresponding variable) can enter the critical section. This reasoning is captured by the conditions of busy-waiting loops. When a process wants to enter the critical section, it simply takes the next "ticket" hence giving priority to the other process.

The program graph corresponding to the first process is quite simple and is presented below (the program graph for the second process is analogous).



Now we can use the Definition 7 to obtain the interleaving of the two processes, which is depicted in Figure 1. Since the result is also a program graph, it can be



$a_{11} : x_1 := x_2 + 1$
$a_{12} : (x_2 = 0) \vee (x_1 < x_2)$
$a_{13} : \neg((x_2 = 0) \vee (x_1 < x_2))$
$a_{14} : x_1 := 0$
$a_{21} : x_2 := x_1 + 1$
$a_{22} : (x_1 = 0) \vee (x_2 < x_1)$
$a_{23} : \neg((x_1 = 0) \vee (x_2 < x_1))$
$a_{24} : x_2 := 0$

Fig. 1. Interleaved program graph.

analysed in our framework.

5 Flow Algebras over Program Graphs

Having defined flow algebras and program graphs, it remains to show how to obtain the analysis results. We shall consider two approaches, namely *MOP* and *MFP*. As already mentioned, these stand for Meet Over all Paths and Maximal Fixed Point, respectively. However, since we take a *join* (least upper bound) to merge information from different paths, in our setting these really mean join over all paths and least fixed point. However, we use the *MOP* and *MFP* acronyms for historical reasons.

We consider the *MOP* solution first, since it is more precise and captures what we would ideally want to compute.

Definition 8. *Given an abstract program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$*, and two sets* $\mathsf{Q}_\circ \subseteq \mathsf{Q}$ *and* $\mathsf{Q}_\bullet \subseteq \mathsf{Q}$ *we are*

interested in

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{\pi \in \mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet)} \mathcal{A}[\![\pi]\!]$$

where

$$\mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \{a_1 a_2 \cdots a_k \mid \exists q_0, q_1, \cdots q_k : \\ q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k, \\ q_0 \in \mathsf{Q}_\circ, q_k \in \mathsf{Q}_\bullet\}$$

and

$$\mathcal{A}[\![a_1 a_2 \cdots a_k]\!] = 1 \otimes \mathcal{A}[\![a_1]\!] \otimes \mathcal{A}[\![a_2]\!] \otimes \cdots \otimes \mathcal{A}[\![a_k]\!]$$

Since the *MOP* solution is not always computable (e.g. for Constant Propagation), one usually uses the *MFP* one which only requires that the lattice satisfies the Ascending Chain Condition and is defined as the least solution to a set of constraints. Let us first introduce those constraints.

Definition 9. *Consider an abstract program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$*. This gives rise to a set* $Analysis_F$ *of constraints*

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq \begin{cases} \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_F & , \text{ if } q \in \mathsf{Q}_\circ \\ \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} & , \text{ if } q \notin \mathsf{Q}_\circ \end{cases}$$

where $q \in \mathsf{Q}$*,* $\mathsf{Q}_\circ \subseteq \mathsf{Q}$*.*

We write $An_F^{\mathsf{Q}_\circ} \models Analysis_F$ whenever $An_F^{\mathsf{Q}_\circ} : \mathsf{Q} \to F$ is a solution to the constraints $Analysis_F$. Now we establish that there is always a least (i.e. best) solution of those constraints.

Lemma 4. *The set of solutions to the constraint system from Definition 9 is a Moore family (i.e. it is closed under* $\bigsqcap$*), which implies the existence of the least solution.*

Definition 10. *We define MFP to be the least solution to the constraint system from Definition 9.*

The following result states the general relationship between *MOP* and *MFP* solutions and shows in which cases the they coincide.

Proposition 1. *Consider the MOP and MFP solutions for an abstract program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *defined over a complete flow algebra* $(F, \oplus, \otimes, 0, 1)$*, then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

If the flow algebra is affine and either $\forall q \in \mathsf{Q} : \mathsf{Path}(\mathsf{Q}_\circ, \{q\}) \neq \emptyset$ *or the flow algebra is strict then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

This is consistent with the previous results, e.g. for Monotone Frameworks, where the *MOP* and *MFP* coincide in case of distributive frameworks and otherwise *MFP* is a safe approximation of *MOP* [9].

6 Galois Connections for Program Graphs

In the current section we show how the generalisation of Galois connections to flow algebras can be used to upper-approximate solutions of the analyses. Namely, consider a flow algebra $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ and a Galois connection (L, α, γ, M). Moreover, let $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ be a flow algebra that is an upper-approximation of the flow algebra over L. We show that whenever we have a solution for an analysis in M then, when concretised, it is an upper-approximation of the solution of an analysis in L. First we state necessary requirements for the analyses. Then we present the results for the *MOP* and *MFP* solutions.

6.1 Upper-approximation of Program Graphs

Since analyses using abstract program graphs are defined in terms of functions specifying effects of different actions, we need to impose conditions on these functions.

Definition 11. *Consider a flow algebra* $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ *that is an upper-approximation of* $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ *by a Galois connection* (L, α, γ, M)*. A program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ *is an upper-approximation of another program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *if*

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) \sqsubseteq_M \mathcal{B}[\![a]\!]$$

It is quite easy to see that this upper-approximation for action implies one for paths.

Lemma 5. *If a program graph* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M)*, then for every path* π *we have that*

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![\pi]\!])$$

Consider a flow algebra $(M, \oplus_M, \otimes_M, 0_M, 1_M)$ induced from $(L, \oplus_L, \otimes_L, 0_L, 1_L)$ by a Galois connection (L, α, γ, M). As in case of flow algebras, we say that a program graph $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ is induced from $(\mathsf{Q}, \Sigma, \to, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ if we change the inequality from Lemma 5 to

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) = \mathcal{B}[\![a]\!]$$

6.2 Preservation of the *MOP* and *MFP* solutions

Now we will investigate what is the relationship between the solutions of an analysis in case of original program graph and its upper-approximation. Again we will first consider the *MOP* solution and then *MFP* one.

MOP We want to show that if we calculate the *MOP* solution of the analysis $\mathcal{B}$ in M and concretise it (using γ), then we will get an upper-approximation of the *MOP* solution of $\mathcal{A}$ in L.

Lemma 6. *If a program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$MOP_L(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \gamma(MOP_M(\mathsf{Q}_\circ, \mathsf{Q}_\bullet))$$

Proof. The result follows from Lemma 5. □

MFP Let us now consider the *MFP* solution. We would like to prove that whenever we have a solution An_M of the constraint system $Analysis_M$ then, when concretised, it also is a solution to the constraint system $Analysis_L$. This is established by the following lemma.

Lemma 7. *If a program graph given by* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$An_M^{\mathsf{Q}_\circ} \models Analysis_M \implies \gamma \circ An_M^{\mathsf{Q}_\circ} \models Analysis_L$$

and in particular

$$MFP_L^{\mathsf{Q}_\circ} \sqsubseteq \gamma \circ MFP_M^{\mathsf{Q}_\circ}$$

Proof. We consider only the case where $q \in \mathsf{Q}_\circ$ *(the other case is analogous). From the assumption we have*

$$\begin{aligned}\gamma \circ An_M^{\mathsf{Q}_\circ} &\sqsupseteq \lambda q.\gamma(\bigoplus\{An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_M)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(1_M)\end{aligned}$$

Now using the definition of upper-approximation of a flow algebra it follows that

$$\begin{aligned}&\lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(1_M)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \gamma(\mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus 1_L\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_L\end{aligned}$$

We also know that every solution $An_L^{\mathsf{Q}_\circ}$ *to the constraints* $Analysis_L$ *must satisfy*

$$An_L^{\mathsf{Q}_\circ} \sqsupseteq \lambda q.\bigoplus\{An_L^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_L$$

and it is clear that $\gamma \circ An_M^{\mathsf{Q}_\circ}$ *is also a solution these constraints.* □

7 Application to the Bakery Algorithm

In this section we use flow algebras and Galois insertions to verify the correctness of the Bakery mutual exclusion algorithm. Although the Bakery algorithm is designed for an arbitrary number of processes, we consider the simpler setting with two processes, as in Example 3. For reader's convenience we recall the pseudo-code of the algorithm:

```
do true ->                                  ||  do true ->
    x1 := x2 + 1;                           ||      x2 := x1 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->           ||      do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip                                ||          skip
    od;                                     ||      od;
    critical section                        ||      critical section
    x1 := 0                                 ||      x2 := 0
od                                          ||  od
```

We want to verify that the algorithm ensures mutual exclusion, which is equivalent to checking whether the state $(3, 3)$ (corresponding to both processes being in the critical section at the same time) is unreachable in the interleaved program graph. First we define the collecting semantics, which tells us the potential values of the variables x_1 and x_2. Since they can never be negative, we take the complete lattice to be the monotone function space over $\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0})$. This gives rise to the flow algebra $\mathcal{C}$ of the form

$$(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}) \to \mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \cup, ;, \lambda ZZ.\emptyset, \lambda ZZ.ZZ)$$

The semantic function is defined as follows

$$\begin{aligned}
\mathcal{T}[\![x_1 := x_2 + 1]\!] &= \lambda ZZ.\{(z_2 + 1, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := x_1 + 1]\!] &= \lambda ZZ.\{(z_1, z_1 + 1) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_1 := 0]\!] &= \lambda ZZ.\{(0, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := 0]\!] &= \lambda ZZ.\{(z_1, 0) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![e]\!] &= \lambda ZZ.\{(z_1, z_2) \mid \mathcal{E}[\![e]\!](z_1, z_2) \wedge (z_1, z_2) \in ZZ\}
\end{aligned}$$

where $\mathcal{E} : \mathit{Expr} \to (\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \to \{\mathit{true}, \mathit{false}\})$ is used for evaluating expressions. Unfortunately, as the values of x_1 and x_2 may grow unboundedly, the underlying transition system of the parallel composition of two processes is infinite. Hence it is not possible to naively use it to verify the algorithm.

Therefore we clearly need to introduce some abstraction. Using our approach we would like to define an analysis that is an upper-approximation of the collecting semantics. This should allow us to compute the result and at the same time guarantee that the analysis is semantically correct. The only remaining challenge is to define a domain that is precise enough to capture the property of interest and then show that the analysis is an upper-approximation of the collecting semantics.

For our purposes it is enough to record when the conditions allowing to enter the critical section (e.g. $(x_2 = 0) \vee (x_1 < x_2)$) are true or false. For that we can

use the Sign Analysis. We take the complete lattice to be the monotone function space over $\mathcal{P}(\mathbb{S} \times \mathbb{S} \times \mathbb{S})$, where $\mathbb{S} = \{-, 0, +\}$. The three components record the signs of variables x_1, x_2 and their difference i.e. $x_1 - x_2$, respectively. We define a Galois connection $(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \alpha, \gamma, \mathcal{P}(\mathbb{S} \times \mathbb{S} \times \mathbb{S}))$ by the extraction function

$$\eta(z_1, z_2) = (sign(z_1), sign(z_2), sign(z_1 - z_2))$$

where

$$sign(z) = \begin{cases} - \text{ if } z < 0 \\ 0 \text{ if } z = 0 \\ + \text{ if } z > 0 \end{cases}$$

Then α and γ are defined by

$$\alpha(ZZ) = \{\eta(z_1, z_2) \mid (z_1, z_2) \in ZZ\}$$
$$\gamma(SSS) = \{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\}$$

for $ZZ \subseteq \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ and $SSS \subseteq \mathbb{S} \times \mathbb{S} \times \mathbb{S}$.

However, note that the set $\mathcal{P}(\mathbb{S} \times \mathbb{S} \times \mathbb{S})$ contains superfluous elements, such as $(0, 0, +)$. Therefore we reduce the domain of the Sign Analysis to the subset that contains only meaningful elements. For that purpose we use the already defined extraction function η. The resulting set $\mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}])$ is defined using

$$\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}] = \{\eta(z_1, z_2) \mid (z_1, z_2) \in \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}\}$$

It is easy to see that

$$\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}] = \left\{ \begin{array}{l} (0, 0, 0), (0, +, -), (+, 0, +), \\ (+, +, 0), (+, +, +), (+, +, -) \end{array} \right\}$$

This gives rise to a Galois insertion (recall Example 2)

$$(\mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}) \to \mathcal{P}(\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}), \alpha', \gamma', \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]) \to \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]))$$

where:

$$\alpha'(f) = \alpha \circ f \circ \gamma$$
$$\gamma'(g) = \gamma \circ g \circ \alpha$$

We next consider the flow algebra $\mathcal{S}$ given by

$$(\mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]) \to \mathcal{P}(\eta[\mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}]), \cup, ;, \lambda SSS.\emptyset, \lambda SSS.SSS)$$

and note that it is induced from the flow algebra $\mathcal{C}$ by the Galois insertion (for details refer to Example 2). Now we can induce transfer functions for the Sign Analysis. As an example let us consider the case of $x_2 := 0$ and calculate

$$\begin{aligned} \mathcal{A}[\![x_2 := 0]\!](SSS) &= \alpha(\mathcal{T}[\![x_2 := 0]\!](\gamma(SSS))) \\ &= \alpha(\mathcal{T}[\![x_2 := 0]\!](\{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\})) \\ &= \alpha(\{(z_1, 0) \mid \eta(z_1, z_2) \in SSS\}) \\ &= \{(s_1, 0, s_1) \mid (s_1, s_2, s) \in SSS\} \end{aligned}$$

Other transfer functions are induced in a similar manner and are omitted.

Clearly the program graph over flow algebra $\mathcal{S}$ is an upper-approximation of the collecting semantics. It follows that the Sign Analysis is semantically correct. Therefore we can safely use it to verify the correctness of the Bakery algorithm. For the actual calculations of the least solution for the analysis problem we use the *Succinct Solver* [13], in particular its latest version [8] that is based on Binary Decision Diagrams [4]. The analysis is expressed in $ALFP$ (i.e. the constraint language of the solver). The result obtained for the node $(3, 3)$ is the empty set, which means that the node is unreachable. Thus the mutual exclusion property is guaranteed.

8 Conclusions

In this paper we presented a general framework that uses program graphs and flow algebras to define analyses. One of our main contributions is the introduction of flow algebras, which are algebraic structures less restrictive than idempotent semirings. Their main advantage and our motivation for introducing them is the ability to directly express the classical analyses, which is clearly not possible when using idempotent semirings. Moreover the presented approach has certain advantages over Monotone Frameworks, such as the ability to handle both sequential and concurrent systems in the same manner. We also define both *MOP* and *MFP* solutions and establish that the classical result of their coincidence in case of distributive analyses carries over to our framework.

Furthermore we investigated how to use Galois connections in this setting. They are a well-known and powerful technique that is often used to "move" from a costly analysis to a less expensive one. Our contribution is the use of Galois connections to upper-approximate and induce flow algebras and program graphs. This allows inducing new analyses such that their semantic correctness is preserved.

Finally we applied our approach to a variant of the Bakery mutual exclusion algorithm. We verified its correctness by moving from a precise, but uncomputable analysis to the one that is both precise enough for our purposes and easily computable. Since the analysis is induced from the collecting semantics by a Galois insertion, we can be sure that the it is semantically correct.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Pearson/Addison Wesley, Boston, MA, USA, second edn. (2007)
2. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551– (2003)
4. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)

5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
7. Droste, M., Kuich, W.: Semirings and formal power series. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, pp. 3–28. Monographs in Theoretical Computer Science. An EATCS Series, Springer Berlin Heidelberg (2009)
8. Filipiuk, P., Nielson, H.R., Nielson, F.: Explicit versus symbolic algorithms for solving ALFP constraints. Electr. Notes Theor. Comput. Sci. 267(2), 15–28 (2010)
9. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Inf. 7, 305–317 (1977)
10. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206 (1973)
11. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
13. Nielson, F., Seidl, H., Nielson, H.R.: A succinct solver for ALFP. Nord. J. Comput. 9(4), 335–372 (2002)
14. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
15. Rosen, B.K.: Monoids for rapid data flow analysis. SIAM J. Comput. 9(1), 159–196 (1980)