A Dataflow Analysis for Comparing and Reordering Predicate Arguments

Yernaux, Gonzague; Vanhoof, Wim

Published in:
Proceedings of the 39th International Conference on Logic Programming

DOI:
10.4204/EPTCS.385.5

Publication date:
2023

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
Yernaux, G & Vanhoof, W 2023, A Dataflow Analysis for Comparing and Reordering Predicate Arguments. in E Pontelli, S Costantini, C Dodaro, S Gaggl, R Calegari, A D'Avila Garcez, F Fabiano, A Mileo, A Russo & F Toni (eds), *Proceedings of the 39th International Conference on Logic Programming: Imperial College London, UK, 9th July 2023 - 15th July 2023.* vol. 385, Electronic Proceedings in Theoretical Computer Science, EPTCS, Open Publishing Association, pp. 41 - 54. https://doi.org/10.4204/EPTCS.385.5

doi:10.1017/xxxxx

A Dataflow Analysis for Comparing and Reordering Predicate Arguments

YERNAUX GONZAGUE
University of Namur, Belgium

VANHOOF WIM
University of Namur, Belgium

Abstract

In this work, which is done in the context of a (moded) logic programming language, we devise a data-flow analysis dedicated to computing what we call argument profiles. Such a profile essentially describes, for each argument of a predicate, its functionality, i.e. the operations in which the argument can be involved during an evaluation of the predicate, as well as how the argument contributes to the consumption and/or construction of data values. While the computed argument profiles can be useful for applications in the context of program understanding (as each profile essentially provides a way to better understand the role of the argument), they more importantly provide a way to discern between arguments in a manner that is more fine-grained than what can be done with other abstract characterizations such as types and modes. This is important for applications where one needs to identify correspondences between the arguments of two or more different predicates that need to be compared, such as during clone detection. Moreover, since a total order can be defined on the abstract domain of profiles, our analysis can be used for rearranging predicate arguments and order them according to their functionality, constituting as such an essential ingredient for predicate normalization techniques.

KEYWORDS: Dataflow analysis, Logic Programming, Argument profiles, Ordering Predicate Arguments, Code Normalization

1 Introduction

When writing code, subroutines (be it methods, procedures, functions or predicates) and their arguments play an important role, as they constitute the main mechanism by which the programmer can make his or her code modular and general and thus applicable in different contexts. While this is true in any language, it is even more so in declarative languages where modularity is often more fine-grained, resulting in lots of small functions and predicates, and where the lack of iterative control structures makes induction-based control (which itself heavily relies on argument manipulation) the rule rather than the exception (Fitting 2002). In this work we consider logic programming and thus predicates as the program's main building blocks.

Understanding the source code of a predicate requires thus understanding the role of the arguments involved, and the data flow relations expressed within the code. If one pursues debugging purposes for instance, statically inferring upon which potential instructions (or, in a logic programming context, atoms) each argument does or does not have influence is crucial to better understand the program at hand (Langevine et al. 2001; Ward and Zedan 2007). While dataflow analysis is a well-known and indispensable ingredient in applications such as code comprehension (Kargén and Shahmehri 2012), compiler optimization (Cooper et al. 2006) and automatic parallelization (Muthukumar et al. 1999), its potential has, to the best of our knowledge, been less explored in applications such as code normalization, anti-unification and clone detection (Pizzolotto and Inoue 2020; Rattan et al. 2013) which is the prime motivation for the current work.

Indeed, when comparing predicate definitions during clone detection or anti-unification, one wants to detect as many (dis)similarities as possible (Yernaux and Vanhoof 2019). It is then often important to consider the right matching between the respective arguments, as the following somewhat contrived example shows. Consider the traditional definition of the `append/3` predicate and another predicate, `concat/3`:

```
append([],L,L).
append([X|Xs],Y,[X|Zs]):- append(Xs,Y,Zs).

concat(L,[],L).
concat([E|Zs],[E|Es],Y):- concat(Zs,Es,Y).
```

Intuitively it is clear that the two predicates define essentially the same ternary relation, where one argument is the concatenation of the two others. The code of the two predicates differs not only in the names of the variables used, but also in the role played by the arguments. Indeed, for an atom append(t_1,t_2,t_3) to succeed, t_3 must be the concatenation of t_1 and t_2 whereas for an atom concat(t_1,t_2,t_3) to succeed, it is t_1 that must be the concatenation of t_2 and t_3. For an analysis to detect that one of these predicates is a "clone" – a textual variant (renaming) of the other *modulo a permutation of the arguments*, it needs to consider potentially all possible argument permutations which adds a non-negligible factor to the complexity of the detection process. In fact, the search for a so-called argument mapping (designating the pairing of corresponding arguments in two predicates) that maximizes the outlined similarity of the involved definitions is one of the key factors rendering a search-based clone detection procedure or, more broadly, the computation of so-called *predicative anti-unification* intractable (Yernaux and Vanhoof 2022). This is especially true when the predicates to be compared are composed of more than a few clauses, since for each suitable argument mapping, there might exist a large number of potential clause mappings that should be explored to find a functional link between the predicates to be compared.

It is not hard to see that the problem of finding a suitable argument mapping can be alleviated by taking adequate abstractions into account. Type- and mode information, for instance, can substantially reduce the number of argument mappings to consider, at least if a sufficient number of arguments are of different type and/or mode. In the example above type information does not really help (as all arguments are supposed to be of the same list type), but using mode information allows to limit the search for corresponding arguments to the subset of input, respectively output arguments of each predicate.

In a more general setting, the question is related to the problem of reordering the arguments in a standard (and preferably unique) way such that arguments playing a similar role (in different predicates) are positioned in similar positions. Ordering arguments is an important aspect of *code normalization*, a process that, generally speaking, aims at restructuring and simplifying code fragments or programs into some kind of *normal* or *canonical* form (Costantini and Provetti 2005; Bruschi et al. 2007) Again, while type and mode information can be used to classify arguments, it is generally not sufficient to sort all of the arguments in a unique way.

In this work, we introduce the notion of an argument profile being an abstract characterization of how that argument is used within the predicate and we devise an analysis capable of computing such profiles. Our approach encompasses, to some extent, type and mode information, but goes further by incorporating into the abstract domain the operations in which the argument participates. While the result of our analysis is not guaranteed to identify each and every argument by a unique value, examples show that it is capable of distinguishing between arguments much more precisely than approaches using only type and mode information.

2 Basic Concepts and Notations

In this paper we consider a simple logic language $\mathcal{L}$ where predicates, clauses, atoms and terms are used and defined in a style similar to that of Prolog. The language is however moded and represents, as such, certain similarities with (a subset of) Mercury (Henderson et al. 1999). We assume given a finite set of variables $\mathcal{V}$, a finite set of functor symbols $\mathcal{F}$ and a finite set of predicate symbols $\mathcal{P}$. As usual variables in $\mathcal{V}$ are strings starting with an uppercase letter while functors and predicates from $\mathcal{F}$, respectively $\mathcal{P}$ are written p/n where p is a string starting with a lowercase letter or symbol called the name of the functor (resp. predicate) and $n \in \mathbb{N}$ its arity, i.e. its number of arguments. We will ease notation by supposing that if a predicate (or functor) p/n exists in the program, then no predicate (or functor) p/m with $m \neq n$ can exist, so that a predicate (or functor) p/m will sometimes simply be referred to as p. The set of terms constructed from $\mathcal{V}$ and $\mathcal{F}$ is denoted $\mathcal{T}$. A term $t \in \mathcal{T}$ is said to be ground if it contains no variables.

A program is defined as a set of predicate definitions, where each predicate is defined by a set of clauses. For simplicity, we will consider only definite clauses, that is each clause is of the form $H \leftarrow B_1, \ldots, B_n$ where H is an atom denoted the head of the clause, and $B_1, \ldots, B_n$ a conjunction of atoms denoting its body. We furthermore assume that the head of a clause contains only variables as arguments (all unifications are made explicit in the body) and that all clauses defining a predicate share the same head. For a predicate p we will use $\mathit{def}(p)$ to denote the set of clauses in its definition and $\mathit{args}(p)$ to denote the sequence of its formal argument variables. With a slight abuse of notation we denote by $\mathit{args}(p)_i$ the ith formal argument of p (i being a number between 1 and the arity of p). For any given program construction c, be it a predicate, a clause, an atom or a clause head, we denote by $\mathit{vars}(c)$ the set of variables occurring in c. We will suppose that each atom in the program is uniquely identified by a natural number from $\mathbb{N}$ that will be referred to as the atom's program point in the program.

We will restrict ourselves to programs that are *directly recursive* to ease the analysis formulation and obtain concrete and efficient results (Debray 1992). Without loss of generality, we will also assume that clause bodies are in some standard, flattened, form in which each atom is either a predicate call having only variables as arguments, or a unification between variables and/or terms in which each term has only an outermost functor (its arguments being variables). We consider our language to be *moded*: each argument appearing in a clause's head is characterized as being either input or output. The argument modes restrict the usage of the predicate in the sense that any call to the predicate must provide a fully instantiated (ground) value for the input arguments, whereas each output argument will be a free variable that is guaranteed to be bound to a ground value upon success of the call. Likewise, unifications are moded as well.

Definition 1
A moded unification is an atom in one of the following forms.

- $V \Rightarrow f(X_1, \ldots, X_n)$, called *deconstruction*, where V is supposed to be input and $X_1, \ldots, X_n$ output. It succeeds if the value bound to V has f/n as an outermost functor in which case it binds $X_1, \ldots, X_n$ to the values figuring in the arguments of f/n.
- $V \Leftarrow f(X_1, \ldots, X_n)$, called *construction*, where V is supposed to be output and $X_1, \ldots, X_n$ input. The construction succeeds if during evaluation $f(X_1, \ldots, X_n)$ is a ground value that can be bound to the free variable V.
- $V \leftrightarrow W$, called *test*, where both V and W are supposed to be input. The test succeeds if both V and W are bound to identical ground values.
- $V := W$, called *assignment*, where V is supposed to be output, and W input. The assignment succeeds if W is bound to a ground value that can be assigned to the free variable V.

Given these constructions and the moded context, our predicates do to some extent resemble what are called *procedures* in Mercury (Henderson et al. 1999).

Example 1
If we represent lists in the usual way, by a functor *nil* representing the empty list and a functor *cons/2* for list construction, the predicate `app/3` below, defined to be used in a mode `app(input,input,output)` realizes the classical ground list concatenation operation in $\mathcal{L}$. The first two arguments are thus supposed to be input, the third one output. The subscript numbers represent the atoms' program points.

$$\begin{array}{lll} app(X,Y,Z) & \leftarrow & X \Rightarrow_1 nil, Z :=_2 Y. \\ app(X,Y,Z) & \leftarrow & X \Rightarrow_3 cons(E,Es), app_4(Es,Y,Zs), Z \Leftarrow_5 cons(E,Zs). \end{array}$$

In the remainder, we will use $\mathcal{A}$ to represent the set of atoms (predicate calls and unifications) as they can occur in the program text, i.e. in the flat form defined above. For an atom $A \in \mathcal{A}$, we denote by $in(A)$ the input arguments of A and by $out(A)$ its output arguments. Note that this only concerns variables, i.e. for any $A \in \mathcal{A}$ we have $in(A) \subseteq vars(A)$ and $out(A) \subseteq vars(A)$. As usual, a substitution is a mapping from variables to terms and applying a substitution θ to a syntactical construct e, written $e\theta$, denotes the construct obtained by simultaneously replacing in e all variables from the domain of θ, denoted $dom(\theta)$, with their corresponding value. Given substitutions θ and σ, their composition $\theta \circ \sigma$ is also written as $\theta\sigma$. A *renaming* $\rho : \mathcal{V} \mapsto \mathcal{V}$ is a special kind of substitution as it is an injective (and idempotent) mapping between variables.

We suppose that programs, when executed, behave in a mode-correct way, meaning that if an instance of an atom (be it a unification or a predicate call) is selected for resolution, the arguments in the atom's input positions are bound to ground values, whereas the arguments in the output positions are unbound variables. To formalise the semantics of our language, we thus introduce the notion of a mode-correct instance.

Definition 2
Let $A \in \mathcal{A}$ be an atom (predicate call or unification). We say that A' is a *mode-correct instance* of A if and only if there exists a substitution θ such that $A' = A\theta$ and

(1) $\forall X \in in(A) : \theta(X)$ is a ground term;
(2) $\forall X \in out(A) : \theta(X)$ is a free variable if $X \in dom(\theta)$.

The semantics of the moded unifications defined above can easily be defined as follows:

Definition 3
Let $U \in \mathcal{A}$ denote a unification and $U\theta$ (for some substitution θ) a mode-correct instance. Then we say that *$U\theta$ succeeds with answer θ'* if and only if the following holds:

- If U is of the form $X \Rightarrow f(Y_1, \ldots, Y_n)$ it holds that $\theta(X) = f(t_1, \ldots, t_n)$ and $\theta' = \{Y_1/t_1, \ldots, Y_n/t_n\}$.
- If U is of the form $X \Leftarrow f(Y_1, \ldots, Y_n)$ it holds that $\theta' = \{X/f(\theta(Y_1), \ldots, \theta(Y_n))\}$.
- If U is of the form $X \leftrightarrow Y$ it holds that $\theta(X) = \theta(Y)$ and $\theta' = \emptyset$.
- If U is of the form $X := Y$ it holds that $\theta' = \{X/\theta(Y)\}$.

The operational semantics of a program is defined in function of a query as usual.

Definition 4
Given a program P, let Q be a query of the form $\leftarrow A_1, \ldots, A_n$. We say that a query Q' *is derived from Q with answer θ* if and only if one of the following conditions holds:

1. A_1 is a mode-correct instance of a unification that succeeds with answer θ, and Q' is the query $\leftarrow (A_2, \ldots, A_n)\theta$.
2. A_1 is a mode-correct instance $p(t_1, \ldots, t_n)$ of the head $H = p(X_1, \ldots, X_n)$ of a (renamed apart) clause $H \leftarrow B_1, \ldots, B_k \in P$ and Q' is the query $\leftarrow (B_1, \ldots, B_k, A_2, \ldots, A_n)\theta$ and $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$.

The above definition is basically equivalent to a traditional SLD-resolution step (with a leftmost selection rule) except for the explicit handling of the (moded) unifications and the limitation to resolving mode-correct instances of atoms only. Next, we can define the notion of a derivation as a sequence of individual derivation steps.

Definition 5
Given a program P and query Q_0. A *derivation* for Q in P is a sequence of queries and substitutions $Q_0 \overset{\theta_0}{\to} Q_1 \overset{\theta_1}{\to} \ldots \overset{\theta_{n-1}}{\to} Q_n$ such that Q_i is derived from Q_{i-1} with answer θ_{i-1} for each $1 \leq i \leq n$. If Q_n is the empty query $\diamond$ then we say that the derivation is *successful* and has associated computed answer substitution $\theta_0\theta_1 \ldots \theta_{n-1}$.

Again, our notion of a derivation is essentially equivalent to an SLD-derivation with a left-to-right selection rule. However, as a consequence of the simple mode system, all computed answers are ground substitutions.

3 Argument and Predicate Profiles

The analysis described in the next section essentially interprets a well-moded logic program and registers the encountered operations into special sets called *interaction sets* that will in the end allow to define a so-called *profile* for each of the predicate's arguments. The key idea of this section is to formalize the values that will be computed and manipulated by our analysis.

First, let us abstract n-ary computations by the dataflow relations that are exhibited between the arguments of a predicate, each dataflow relation being annotated by the set of operations that participate in the relation. Among the operations of interest are the basic unification operators defined by the set B as follows:

$$B = \{:=, \leftrightarrow\} \cup \bigcup_{f \in \mathcal{F}} \{\Leftarrow_f, \Rightarrow_f\}$$

For a given argument, we will represent a single dataflow relation it participates in by means of an *o-set*, the latter being essentially a tuple (o, j) in which o represents a subset of operations (from a given set of admissible operations, like B above) and j a natural number representing the position of one of the (other) arguments. More formally:

Definition 6
Given a set of operations S, we define the set of *o-sets over* S as

$$OS(S) = \{(o, j) \mid o \in \mathbb{P}(S) \text{ and } j \in \mathbb{N}\}$$

In general, an argument participates in more than one dataflow relation, relating it to several other arguments (each time by means of a set of operations). To represent such a *set* of dataflow relations, we introduce the notion of an *argument profile*. Intuitively, an argument profile for the i'th argument of p/n denotes a set of dataflow dependencies with some of the other arguments of p, where each dependency is represented – through an o-set – by the set of operations linking both arguments. Formally, we define the notion of an argument profile for an n-ary operation as follows:

Definition 7
Given a set of operations S and $n \in \mathbb{N}$, we define an argument profile for an n-ary operation with respect to S as a set $A \subseteq OS(S)$ where for each $(o, j) \in A$ we have that $j \in \{1, \dots, n\}$. We will use $AP_n(S)$ to represent the set of all possible argument profiles for an n-ary operation with respect to S.

Example 2
The following is an argument profile: $\{(\{\Rightarrow_{cons}, :=\}, 2), (\{\Leftarrow_{cons}\}, 3)\}$. It represents the fact that the concerned argument is involved through a deconstruction in a list, and an assignment, with the value of the argument in position 2. It similarly helps building the argument in position 3 by a list construction atom.

The above definitions are fine as long as we restrict ourselves to using operations from a fixed set of operations such as B. However, it is worthwhile to include among the allowed operations also those operations defined (by means of predicates) in the program itself. We will not include the predicates as such in the set of admissible operations as it would make the domain too dependent on the names chosen for the predicates at hand. Rather, we will use abstractions of these predicates – notably those abstractions our analysis aims to compute. As such, the basic idea is to represent an n-ary operation (or predicate) by means of a term $\psi(\alpha_1, \dots, \alpha_n)$ where the α are argument profiles. A special term $\psi_\perp$ is introduced in order to represent an operation for which no argument profiles are known; in the analysis it will be used to represent direct recursive calls. Since the ψ-terms use argument profiles that themselves can contain ψ-terms, we define the set of all possible abstract operations as the least fixed point of the following operator R:

Definition 8
Given a set of operations S, we define

$$R(S) = B \cup \{\psi_\perp\} \cup \bigcup_{n \in \mathbb{N}_0} \{\psi(\alpha_1, \ldots, \alpha_n) \mid \alpha_i \in AP_n(S)\}$$

While $\mathit{lfp}(R)$ contains some infinite terms, all terms created by our analysis will be of finite size, as will be made clear further down. In the following we use AP_n to refer to the set of all possible argument profiles for an n-ary operation with respect to $OS(\mathit{lfp}(R))$. We will refer to the elements of $\mathit{lfp}(R)$ in which a ψ appears as *ψ-based operations.*

In order to obtain argument profiles, the analysis will compute data flow relations within a predicate, annotated with the operations that are encountered upon establishing the relation. We thus define an *interaction* as being the association of an input variable and an output variable with a set of operations and the program points these operations are occurring at. Formally:

Definition 9
Let p be a predicate in a program P. An *interaction in p* is a mapping $\mathit{vars}(p) \times \mathit{vars}(p) \mapsto \mathbb{P}(\mathit{lfp}(R) \times \mathbb{N})$. Notation-wise, we will typically write $V \overset{O}{\rightsquigarrow} \hat{V}$ to represent an interaction between a variable V and another variable $\hat{V}$ through a set $O \subset \mathit{lfp}(R) \times \mathbb{N}$.

In order not to overload our notation, when writing interactions, we will usually drop the program points and consider the sets of operations in an interaction to be a multiset $O \subset \mathit{lfp}(R)$. We will thus allow doubles in the set, assuming they are operations implemented by atoms located at different program points. We will only occasionally include program points explicitly when needed in order to explicitly distinguish between identical operations coming from different atoms.

An important characteristic of the set of interactions describing a predicate is that for each pair of variables, there is at most a single interaction between these variables present in the set. Another characteristic is the fact that for any interaction $V \overset{O}{\rightsquigarrow} \hat{V}$ it holds that $\hat{V}$ cannot be an input argument, since mode-correct input arguments cannot be constructed by computations in a predicate's body. V does not have such a limitation, as long as V and $\hat{V}$ are distinct. More formally:

Definition 10
For a predicate p, we call a *well-defined interaction set* for p a set ϕ of interactions in p such that for all $V, \hat{V} \in \mathit{vars}(p)$ it holds that if there exists $V \overset{O}{\rightsquigarrow} \hat{V} \in \phi$ for some O, then the following conditions all hold:

1. $V \neq \hat{V}$;
2. $\nexists V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi : O' \neq O$;
3. $\hat{V} \in \mathit{args}(p) \Rightarrow \hat{V}$ is an output argument.

We will use ISet_p to denote the set of all well-defined interaction sets for a given predicate p. In case p is clear from the context, we will use the shorter notation ISet. Now we define the following quasi-order allowing to organize ISet_p in a lattice.

Definition 11
Let p be a predicate. For $\phi_1, \phi_2 \in \mathit{ISet}_p$ we say that ϕ_1 is more precise than ϕ_2, denoted $\phi_1 \sqsubseteq \phi_2$, if and only if $\forall V \overset{O}{\rightsquigarrow} \hat{V} \in \phi_1 : \exists V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi_2$ such that $O \subseteq O'$.

That is, $\phi_1 \sqsubseteq \phi_2$ when each interaction appearing in ϕ_1 labeled by an operation set O is matched by an interaction in ϕ_2 that is labeled by an operation set being a superset of O, and ϕ_2 may contain interactions involving pairs of variables that are not linked by an interaction in ϕ_1. We now define the following operator.

Definition 12
For a predicate p, let $\phi \in \mathit{ISet}_p$ and let $V \overset{O}{\rightsquigarrow} \hat{V}$ be an interaction for p. Then we define

$$(V \overset{O}{\rightsquigarrow} \hat{V}) \sqcup \phi = \begin{cases} \{V \overset{O}{\rightsquigarrow} \hat{V}\} \cup \phi & \text{if } \nexists (V \overset{O'}{\rightsquigarrow} \hat{V}) \in \phi \text{ for some } O' \\ (\phi \setminus \{V \overset{O'}{\rightsquigarrow} \hat{V}\}) \cup \{V \overset{O \cup O'}{\rightsquigarrow} \hat{V}\} & \text{otherwise} \end{cases}$$

Note that adding an interaction to a well-defined interaction set results in a well-defined interaction set. It can also be easily seen that when constructing a well-defined interaction set, the order in which the individual interactions are added has no influence on the final result. Consequently, we can extend the $\sqcup$ operator such that it merges two well-defined interaction sets:

Definition 13
Let ϕ and ϕ' be well-defined interaction sets for a predicate p. Then we define $\phi \sqcup \phi'$ as the following well-defined interaction set: $\phi \sqcup \phi' = \bigsqcup_{V \overset{O}{\rightsquigarrow} \hat{V} \in \phi} (V \overset{O}{\rightsquigarrow} \hat{V}) \sqcup \phi'$.

Proposition 1
For a predicate p, $(\mathit{ISet}_p, \sqcup)$ is a join semi-lattice.

The induced partial order, namely $\sqsubseteq$, is such that $\phi \sqsubseteq \phi'$ if and only if $\phi \sqcup \phi' = \phi'$, so that we get a partially ordered set $(\mathit{ISet}_p, \sqsubseteq)$ in which each subset $\{\phi_1, \ldots, \phi_n\}$ has a least upper bound, namely $\sqcup\{\phi_1, \ldots, \phi_n\}$. The partially ordered set has a minimal element, namely the empty set $\{\}$ which we will refer to by $\bot$ as it is a unit for the join operator: $\forall \phi \in \mathit{ISet}_p : \bot \sqcup \phi = \phi \sqcup \bot = \phi$. The maximal element $\top_p \in \mathit{ISet}_p$ is the set containing all possible interactions between each argument and all the (other) output arguments.

The goal of our analysis is to compute, for each predicate p in a given program P, a well-defined interaction set for p. This element of ISet_p will be such that it only reflects the interactions between variables $V, \hat{V}$ such that $V, \hat{V} \in \mathit{args}(p)$. Such an element is what we will call a *predicate profile.*

Definition 14
Given a program P and a predicate p defined therein. A *predicate profile* for p is a well-defined interaction set ϕ of interactions in p such that for all $V \overset{O'}{\rightsquigarrow} \hat{V} \in \phi$ we have that V and $\hat{V}$ are formal arguments of p, that is $\{V, \hat{V}\} \subseteq \mathit{args}(p)$.

We can "decompose" a predicate profile into individual argument profiles as follows:

Definition 15

Given a program P, a predicate p in P, and a predicate profile ϕ for p, we define the argument profile of the i'th argument of p with respect to ϕ as the following set of o-sets:

$$\alpha_i = \{(O, j) \mid V_i \overset{O}{\rightsquigarrow} V_j \in \phi\}$$

where $V_i = \mathit{args}(p)_i$ and $V_j = \mathit{args}(p)_j$. Moreover, we define the *computed argument profile* of p with respect to ϕ as the sequence $\langle \alpha_1, \ldots, \alpha_n \rangle$.

Recall that, based on such computed argument profiles, our objective is to *reorder* the predicate arguments, preferably in a unique way. As a first observation, note that it is not hard to define *a* total order on AP as the following example illustrates.

Example 3
For an argument profile $\alpha \in AP$, let us define the *features of* α as the vector $(\#\alpha, o, m, s, r, c, d)$ with o the total number of operations contained in α, r the number of ψ-based operations in it, c, a, d its number of constructions, assignments and deconstructions respectively. Now let α_1 and α_2 be argument profiles with respective tuples t_1 and t_2. We define the total order $\leq$ such that

$$\alpha_1 \leq \alpha_2 \quad \Leftrightarrow \quad t_1 - t_2 = (0) \vee \text{the first non-zero dimension in } t_1 - t_2 \text{ is positive}$$

While the order defined in Example 3 is somewhat arbitrary and not necessarily capable of producing a *unique* order, its definition is independent of the analyzed program. In the following section, we construct our analysis that takes a total order $\leq$ on AP as a parameter. Given such an order $\leq$, for a predicate p with some profile ϕ, we will use $\mathit{opr}(\phi)$ to represent a profile of p ordered by $\leq$ with respect to ϕ.

Definition 16
Given a predicate p/n, a profile ϕ and a total order $\leq$. Let $\langle \alpha_1, \ldots, \alpha_n \rangle$ be the argument profile of p with respect to ϕ. Then we define the *ordered profile* of p with respect to ϕ as a permutation $\langle \alpha'_1, \ldots, \alpha'_n \rangle$ of $\langle \alpha_1, \ldots, \alpha_n \rangle$ such that $\alpha_i \leq \alpha_{i+1}$ for all $1 \leq i < n$.

4 A Dataflow Analysis Computing Argument Profiles

The analysis will basically compute what we call an *environment* which is a mapping from predicates to well-defined interaction sets that represent the already computed interactions between the predicate's formal arguments. We will use the symbol $\Phi : \mathcal{P} \mapsto \mathit{ISet}$ to represent such an environment. The analysis is defined by induction on the syntactic structure of the program's predicates. We start by defining the analysis of an individual atom. It basically incorporates the operations of interest into interactions involving local variables as well as arguments. The analysis is parametrized by the current environment Φ and a total order $\leq$ capable of ordering a predicate profile ϕ into $\mathit{opr}(\phi)$.

Definition 17
Let P be a program of interest. The atomic analysis function $\mathbb{A} : \mathcal{A} \mapsto (\mathcal{P} \mapsto \mathit{ISet}) \mapsto \mathit{ISet}$ is defined as the function that returns, given an atom A and an environment Φ, a set of interactions composed by those operations from $\mathit{lfp}(R)$ that are found occurring in A:

$$\mathbb{A}[\![V \Rightarrow f(Y_1, \ldots, Y_n)]\!]\Phi = \bigsqcup_{i \in 1..n} \{V \overset{\{\Rightarrow_f\}}{\rightsquigarrow} Y_i\}$$

$$\mathbb{A}[\![V \Leftarrow f(Y_1,\ldots,Y_n)]\!]\Phi = \bigsqcup_{i\in 1..n} \{Y_i \overset{\{\Leftarrow_f\}}{\rightsquigarrow} V\}$$

$$\mathbb{A}[\![V := W]\!]\Phi = \{W \overset{\{:=\}}{\rightsquigarrow} V\}$$
$$\mathbb{A}[\![V \leftrightarrow W]\!]\Phi = \{\}$$
$$\mathbb{A}[\![q(Y_1,\ldots,Y_m)]\!]\Phi = \Phi(q)\rho \sqcup \phi_q$$

where $\rho = \{args(q)_1/Y_1,\ldots, args(q)_m/Y_m\}$

and $\phi_q = \left\{Y_i \overset{o}{\rightsquigarrow} Y_j \mid Y_i \in in(q(Y_1,\ldots,Y_m)), Y_j \in out(q(Y_1,\ldots,Y_m))\right\}$

in which $o = \begin{cases} \psi_\perp & \text{if it is a directly recursive call} \\ \psi(opr(\Phi(p))) & \text{otherwise}\end{cases}$

In Definition 17 above, we apply a renaming ρ to a set of interactions $\Phi(q)$, which consists in replacing each variable V from $dom(\rho)$ occurring in $\Phi(q)$ by $\rho(V)$. Using $opr(\Phi(p))$ allows the ψ-based operations occurring in an argument profile to describe atoms based on similar operations by means of normalized values. For instance, as will be made clear later on, whether a predicate makes a call to $app/3$ or to a variant of it where some arguments are swapped, the resulting ψ-based operation will be the same.

Example 4
The following are applications of our function $\mathbb{A}$ on atoms that appear in the predicate *app* from Example 1. We consider given an environment Φ_0 that maps *app* on $\perp$.

$$\mathbb{A}[\![X \Rightarrow cons(E,Es)]\!]\Phi_0 = \{X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} E, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} Es\}$$
$$\mathbb{A}[\![Z \Leftarrow cons(E,Zs)]\!]\Phi_0 = \{E \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z, Zs \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z\}$$
$$\mathbb{A}[\![app(Es,Y,Zs)]\!]\Phi_0 = \{Es \overset{\{\psi_\perp\}}{\rightsquigarrow} Zs, Y \overset{\{\psi_\perp\}}{\rightsquigarrow} Zs\}$$

Extending the analysis function to clauses is relatively straightforward as it suffices to analyze each of the body atoms, joining the results using $\sqcup$. However, we need to include a transitive closure operator that allows to *combine* the interactions resulting from the analysis of the individual atoms such that the resulting interactions represent – where possible – data flow between arguments rather than involving local variables.

Definition 18
Let $p \in \mathcal{P}$ and $\phi \in ISet_p$. Let $T : ISet \mapsto ISet$ denote the following operator

$$T(\phi) = \{X \overset{O\cup O'}{\rightsquigarrow} Z \mid X \overset{O}{\rightsquigarrow} Y, Y \overset{O'}{\rightsquigarrow} Z \in \phi \text{ for some distinct } X,Y,Z \in \mathcal{V}\}$$

and let $cl_T(\phi)$ denote the transitive closure of T on ϕ, that is the smallest relation on ϕ that contains T and is transitive. Then the *projection of* ϕ *onto the arguments of* p is denoted by $\pi_p(\phi)$ and defined as

$$\pi_p(\phi) = \{X \overset{O}{\rightsquigarrow} Y \in cl_T(\phi) \mid X,Y \in args(p)\}.$$

For a given $\phi \in ISet$, the transitive closure $cl_T(\phi)$ can always be computed by merging into ϕ those interactions that can be seen as *transitive interactions*, i.e. interactions that concern three different variables X,Y,Z in the way described in the Definition above.

The number of these transitive interactions is inevitably finite, being proportional to the number of combinations among a finite number of variables.

The analysis of a complete program consists in repeatedly analyzing each and every clause of the program with respect to the current environment, computing as such an updated environment that incorporates the results of the current analysis round.

Definition 19
Let P be a program and $p \in P$ a predicate of interest. The predicate analysis function $\mathbb{S} : \mathcal{P} \mapsto (\mathcal{P} \mapsto \mathit{ISet}) \mapsto \mathit{ISet}$ is defined as the function that returns, given a predicate p and an environment Φ, a well-defined interaction set for p:

$$\mathbb{S}[\![p]\!]\Phi = \bigsqcup_{h \leftarrow a_1,\ldots,a_n \in \mathit{def}(p)} \pi_p\Big(\bigsqcup_{i \in 1\ldots n} \mathbb{A}[\![a_i]\!]\Phi\Big)$$

Note the effect of the different join operations. First, the interaction sets resulting from the analysis of the individual atoms in a clause body are combined (using the innermost join). The outermost join combines the interaction sets resulting from the different clauses, after projection, into a single interaction set. The projection onto the arguments of the predicate is important, as it avoids the construction of spurious interactions caused by the same local variable that might be used in different clauses. The fact that local variables are ignored in the result of the formula above is no limitation, since the operator $\mathbb{S}$ is used below to compute the successive environments, and our analysis uses the environment solely for exploiting the interactions among arguments.

Example 5
Let us consider again the predicate *app* from Example 1. A round of our analysis for *app* is partially depicted in Example 4, its complete result being:

$$\mathbb{S}[\![\mathit{app}]\!]\Phi_0 = \{Y \overset{\{:=,\psi_\perp\}}{\rightsquigarrow} Z, X \overset{\{\Rightarrow_{cons},\psi_\perp \Leftarrow_{cons}\}}{\rightsquigarrow} Z\}$$

which corresponds to the projection on X, Y and Z of the computed interactions $\{Y \overset{\{:=\}}{\rightsquigarrow} Z, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} E, X \overset{\{\Rightarrow_{cons}\}}{\rightsquigarrow} Es, X \overset{\{\psi_\perp\}}{\rightsquigarrow} Z, Y \overset{\{\psi_\perp\}}{\rightsquigarrow} Z, E \overset{\{\Leftarrow_{cons}\}}{\rightsquigarrow} Z, Zs \overset{\{\Leftarrow_{cons}\}Z}{\rightsquigarrow}\}$.

Now, to analyze a program from scratch, we start from an initial environment Φ_0 in which each predicate is associated to an initial interaction set $\perp$. The predicates are subsequently analyzed according to their position in the program's call graph in a bottom-up manner, that is prioritizing those predicates that contain no calls to predicates except maybe themselves or predicates that have previously been analyzed. We will denote by $\mathit{leafs}(P)$ the set of such *eligible* predicates in a program P. Each time a predicate's analysis reaches a fixpoint, the analysis proceeds to the next eligible predicate. The process is repeated until every predicate has been considered. It is depicted in Algorithm 1.

Example 6
Let us resume the analysis of *app*/3 started in Examples 4 and 5, where we obtained an environment value, say Φ_1, after one analysis round. A second round of the analysis will only differ in the handling of the atom $\mathit{app}(Es, Y, Zs)$:

$$\mathbb{A}[\![\mathit{app}(Es, Y, Zs)]\!]\Phi_1 \quad = \quad \{Y \overset{\{:=,\psi_\perp\}}{\rightsquigarrow} Zs, Es \overset{\{\Rightarrow_{cons},\Leftarrow_{cons},\psi_\perp\}}{\rightsquigarrow} Z\}$$

Algorithm 1 Analyzing a program P

```
$PS \leftarrow P, i \leftarrow 0, \Phi_0 \leftarrow \bigcup_{p \in P} \{ (p, \bot) \}$
while $leafs(PS) \neq \emptyset$ do
    select $p \in leafs(PS)$
    while $(\mathcal{S}[\![p]\!]\Phi_i)(p) \neq \Phi_i(p)$ do
        $\Phi_{i+1} \leftarrow \mathcal{S}[\![p]\!]\Phi_i$
        $PS \leftarrow PS \setminus \{p\}$
        $i \leftarrow i + 1$
```

After merging and projection on the arguments, we obtain Φ_2 such that

$$\Phi_2(app) \quad = \quad \{X \overset{\{\Rightarrow_{cons}, \Leftarrow_{cons}, \psi_\bot\}}{\rightsquigarrow} Z, Y \overset{\{\Leftarrow_{cons}, :=, \psi_\bot\}}{\rightsquigarrow} Z\}$$

where the $\Leftarrow_{cons}$ operation linking Y to Z is obtained by the fact that we have both $Y \overset{\{:=\}}{\rightsquigarrow} Zs$ and $Zs \overset{\Leftarrow_{cons}}{\rightsquigarrow} Z$ in the computed interactions set. Any subsequent analysis round would not alter this environment, so that the analysis is finished for app.

Let us now consider that our program is also constituted of a moded version of the $concat/3$ predicate introduced in Section 1:

$$\begin{array}{lcl} concat(A,B,C) & \leftarrow & B \Rightarrow_6 nil, A :=_7 C. \\ concat(A,B,C) & \leftarrow & B \Rightarrow_8 cons(I,Is), concat_9(As,Is,C), A \Leftarrow_{10} cons(I,As). \end{array}$$

Analyzing *concat* yields the interactions $\{B \overset{\{\Rightarrow_{cons}, \Leftarrow_{cons}, \psi_\bot\}}{\rightsquigarrow} A, C \overset{\{\Leftarrow_{cons}, :=, \psi_\bot\}}{\rightsquigarrow} A\}$. Now using $\leq$, the ordered profiles of both predicates are one and the same, namely $\langle\{(\{\Rightarrow_{cons}, \Leftarrow_{cons}, \psi_\bot\}, 2)\}, \{(\{:=, \psi_\bot, \Leftarrow_{cons}\}, 2)\}\rangle$ which corresponds to the respective profiles of X/B, Y/C and Z/A. In other words, reordering the arguments according to $\leq$ leaves *app* untouched but transforms $concat(A,B,C)$ into $concat(B,C,A)$.

The predicate calls in the example above being recursive calls, we introduce the following example to illustrate the case where a predicate makes calls to other predicates.

Example 7
Let us extend Example 6 with the *double append* operation embodied by $dapp/4$:

$$dapp(L1, L2, L3, L4) \quad \leftarrow \quad app_{11}(L1, L2, L12), concat_{12}(L4, L12, L3).$$

The analysis finds the following final interaction set for *dapp*:

$$\left\{ \begin{array}{l} L1 \overset{\{\Rightarrow_{cons}(3), \Leftarrow_{cons}(5), \psi_\bot(4), \psi_a(11), \Rightarrow_{cons}(8), \Leftarrow_{cons}(10), \psi_\bot(9), \psi_a(12)\}}{\rightsquigarrow} L4, \\ L2 \overset{\{:=(2), \psi_\bot(4), \Leftarrow_{cons}(5), \psi_a(11), \Rightarrow_{cons}(8), \Leftarrow_{cons}(10), \psi_\bot(9), \psi_a(12)\}}{\rightsquigarrow} L4, \\ L3 \overset{\{:=(7), \psi_\bot(9), \Leftarrow_{cons}(10), \psi_a\}}{\rightsquigarrow} L4 \end{array} \right\}$$

where $\psi_a = \psi(\{(\{\Rightarrow_{cons}, \Leftarrow_{cons}, \psi_\bot\}, 2)\}, \{(\{:=, \psi_\bot, \Leftarrow_{cons}\}, 2)\})$ and where the program points have been made explicit when applicable.

The example shows that our analysis allows to entirely distinguish the four arguments of the double append operation, whereas type- and mode information alone would not have made a distinction among the first three arguments. Having these profiles for different arguments allows to uniquely order these by using an appropriate $\leq$ operator and, hence, to match $dapp/4$ with predicates that implement the same functionality differently.

The analysis reaches an interaction set fixpoint for each examined predicate.

Proposition 2
The sequence (Φ_n) as defined by Algorithm 1 is convergent.

The interested reader is referred to Appendix B for the proof of Proposition 2, and to Appendix C for a preliminary result on the analysis soundness. In a likewise manner, Appendix D develops a preliminary result on its worst-case time complexity.

5 Conclusions and Future Work

This work aims to develop a tractable process for profiling predicate arguments and normalizing their order of apparition in a prototypical Mercury-like language. Our analysis essentially computes a high-level abstraction of program derivations, called *interactions.* Although a normalization procedure already existed for Mercury (Degrave and Vanhoof 2008), it focused on normalizing clause bodies and did not address predicate arguments.

Our approach to code normalization revolves around the search of an ordering among predicate arguments. Central to this technique is the research for an *ideal* ordering of the arguments, i.e. a total order $\leq$ that allows to sort arguments in a non-ambiguous, unique way, at least in the context of a single program. While we have introduced a first working, but rather arbitrary, example of such an order based on argument profiles metrics, it is our belief that more precise or application-tailored orderings could be found to enhance the analysis output in concrete situations. In particular, identifying the situations in which an order is to be preferred over other incarnations, is left as future work.

Having a normal form for programs is recognized as an important step in several applications, one of interest being a clone detection scheme, where recognizing a couple of similar predicates implies finding a mapping of clauses and a mapping of arguments among the predicates such that two clauses, or arguments, in the mapping play similar roles in the predicate's definition. The problem, which is intractable in general, becomes radically more manageable if a quadratic approximation is found for one of the two interleaved matching problems (Yernaux and Vanhoof 2022). We intend to explore the use of our analysis for computing a matching of arguments in this context.

Program comprehension is a rising research field in which all aspects of dataflow information constitute useful pieces of information. Program slicing, for example, is a way of extracting the computations in which a given (set of) argument(s) plays a prominent role (Ward and Zedan 2007). Interestingly, what we achieve by computing argument profiles resembles the extraction of such program slices. In existing program slicing techniques however, the computed slices are actual parts of the considered program (Ward and Zedan 2007), whereas our profiles rather constitute abstract representations of data flow information. Moreover, while an argument profile typically exhibits the details of the operations (be it unifications or calls to predicates) that involve the argument, the program portions obtained by means of slicing do not carry any *interpretation* of the program, as the slices' purpose is to represent the part of the program that might be of interest (Szilágyi et al. 2002). As an example, consider a predicate in which all of the arguments are somehow participating in every single atom but in different manners. The slices for the different arguments then systematically come down to the whole predicate definition. In contrast, our argument profiles contain finer-grained distinctions, allowing to identify which operations involve which arguments, as well as specific links between input and output arguments – but abstracting from the order in which the involved atoms are executed. We therefore believe our approach to be complementary to program slicing and to constitute a new step towards better understanding links between arguments and, hence, deriving useful information about the operations hidden in a predicate definition.

Other analyses addressing program comprehension or security concerns by studying interactions among variables could benefit from our method, some examples being feature analysis, trace analysis and taint analysis (Eisenbarth et al. 2001; Cornelissen et al. 2009).

References

Bruschi, D., Martignoni, L., and Monga, M. 2007. Code normalization for self-mutating malware. *IEEE Security & Privacy*, *5*, 2, 46–54.

Cooper, K. D., Harvey, T. J., and Kennedy, K. An empirical study of iterative data-flow analysis. In *2006 15th International Conference on Computing* 2006, pp. 266–276.

Cornelissen, B., Zaidman, A., Deursen, A., Moonen, L., and Koschke, R. 2009. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, *35*, 684 – 702.

Costantini, S. and Provetti, A. 2005. Normal forms for answer sets programming. *Theory and Practice of Logic Programming*, *5*.

Debray, S. K. 1992. Efficient dataflow analysis of logic programs. *J. ACM*, *39*, 4, 949–984.

Degrave, F. and Vanhoof, W. Towards a normal form for mercury programs. In King, A., editor, *Logic-Based Program Synthesis and Transformation* 2008, pp. 43–58. Springer.

Eisenbarth, T., Koschke, R., and Simon, D. 2001. Aiding program comprehension by static and dynamic feature analysis.

Fitting, M. 2002. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, *278*, 1, 25 – 51. Mathematical Foundations of Programming Semantics 1996.

Henderson, F., Conway, T., Somogyi, Z., Schachte, P., Taylor, S., and Speirs, C. 1999. The mercury language reference manual.

Kargén, U. and Shahmehri, N. Inputtracer: A data-flow analysis tool for manual program comprehension of x86 binaries. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation* 2012, pp. 138–143.

Langevine, L., Deransart, P., Ducasse, M., and Jahier, E. 2001. Tracing execution of clp(fd) programs : A trace model and an experimental validation environment.

Muthukumar, K., Bueno, F., García de la Banda, M., and Hermenegildo, M. 1999. Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism. *The Journal of Logic Programming*, *38*, 2, 165–218.

Pizzolotto, D. and Inoue, K. Blanker: A refactor-oriented cloned source code normalizer. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)* 2020, pp. 22–25.

Rattan, D., Bhatia, R., and Singh, M. 2013. Software clone detection: A systematic review. *Information and Software Technology*, *55*, 7, 1165–1199.

Szilágyi, G., Gyimóthy, T., and Maluszynski, J. 2002. Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, *9*, 41–65.

Ward, M. and Zedan, H. 2007. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, *29*.

Yernaux, G. and Vanhoof, W. 2019. Anti-unification in Constraint Logic Programming. *Theory and Practice of Logic Programming*, *19*, 5-6, 773–789.

Yernaux, G. and Vanhoof, W. On detecting semantic clones in constraint logic programs. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)* 2022, pp. 32–38.

Appendices

A. Proof of Proposition 1

Proof
We need to prove that for a predicate p, the $\sqcup : ISet_p \times ISet_p \mapsto ISet_p$ operation is idempotent, associative and commutative. This follows directly from the definition of $\sqcup$ (being essentially a union operation on sets of interactions and possibly on sets of operations) and the fact that the union operator on sets is itself idempotent, associative, and commutative. □

B. Proof of Proposition 2

Proof
First note that by construction, the sequence of computed environments $\Phi_0, \Phi_1, \ldots$ is such that $\forall i \in \mathbb{N}_0$, either $\Phi_i = \Phi_{i-1}$ and then Φ_i is the fixpoint of the sequence, or there exists $p \in \mathcal{P}$ such that $\Phi_i(p) \neq \Phi_{i-1}(p)$. In that case, the only possibilities are that

- $\Phi_i(p) \supset \Phi_{i-1}(p)$, due to a new interaction being discovered during the iteration, and/or
- $\exists V \overset{O_1}{\rightsquigarrow} \hat{V} \in \Phi_i(p), V \overset{O_2}{\rightsquigarrow} \hat{V} \in \Phi_{i-1}(p) : O_1 \neq O_2$. This can only happen if a new operation is added to an existing interaction, or if a ψ-based operation is replaced by a different ψ-based operation.

Now, for a predicate p/n, the number of interactions in $\Phi_i(p)$ (for any i) is limited by the number of pairs of (possibly interacting) arguments, which is of the order $O(n^2)$. Likewise, the set of operations labelling an interaction is necessarily finite, as its size is limited by the number of program points in the program. What remains to be shown, is that for an operation (a predicate call, say to some predicate q/m) at a given program point, there is no infinite succession of different ψ-based operations representing this operation. Now, this could only happen if the called predicate q/m was itself re-analysed between analysis rounds of p. This is excluded, as we restricted programs to direct-recursive programs only, and our analysis analyses predicates bottom-up in the call-graph such that when a predicate is analysed that is calling q/m, the analysis results for q/m are definitely known and hence the ψ-based operation representing this call will always be the same (some abstract profile $\psi(\alpha_1, \ldots, \alpha_m)$ or $\psi_\perp$ in case the call is a direct recursive call, i.e. $p = q$). □

C. Towards a proof of soundness

To prove that our analysis is sound, we use the following development. First, recall that the analysis terminates as a consequence of Proposition 2. Now for a given query targeted on a given predicate, consider a successful derivation exhibiting *de facto* the construction of some of the arguments' values by the use of other arguments. To link the results of our analysis to these operational semantics we first need to widen said semantics by the definition and use of what is often called collecting semantics. In our case the collecting semantics should explicit the fact that, in a concrete derivation, some variable V participates in the construction of the value of some other variable W. First, we will extend our notation capabilities by defining the concept of argument positions in an instantiated atom.

Definition 20
Let A be a mode-correct instance of an atom. We denote by $pos(A) \subset \mathbb{N}_0$ the set of valid positions in A and for $i \in pos(A)$ we denote by $A[i]$ the ith *position* in A, in the following sense:

- if $A = p(t_1, \ldots, t_n)$ then $pos(A) = 1..n$ and $\forall i \in pos(A) : A[i] = t_i$;
- if $A = t \Rightarrow f(t_1, \ldots, t_n)$ or $A = t \Leftarrow f(t_1, \ldots, t_n)$ then $pos(A) = 1..n+1$, $A[1] = t$ and $\forall i \in 2..n+1 : A[i] = t_{i-1}$;
- if $A = t_1 \leftrightarrow t_2$ or $A = t_1 := t_2$ then $pos(A) = \{1, 2\}$, $A[1] = t_1$ and $A[2] = t_2$.

Next, we devise the concept of *mark* which is essentially a sequence of argument positions.

Definition 21
Let $(A_i)_{i \in 1..n}$ be a set of mode-correct instances of atoms. A *mark on* $(A_i)_{i \in 1..n}$ is a sequence $A^1[j^1] \rightharpoonup \cdots \rightharpoonup A^k[j^k]$ such that $\forall i \in 1..k : A^i \in (A_i)_{i \in 1..n} \wedge j^k \in pos(A^i)$.

In the following definition we denote, for a query Q, by $A_{|Q}$ the set of atoms that appear in Q, and we denote by $\mathrm{op}(A)$ the underlying operation of an atom $A \in \mathcal{A}$ (that is, the underlying element from $B \cup \mathcal{P}$). We use a mark scoped in a given derivation and defined so as to exhibit a path between two arguments.

Definition 22
Let $\delta = Q_0 \overset{\theta_0}{\to} Q_1 \overset{\theta_1}{\to} \ldots \overset{\theta_{n-1}}{\to} Q_n$ be a successful derivation and V, W variables appearing at least once therein. Let Q_k denote the last query in δ in which V appears and Q_l the same for W. We say that a mark $A_1[j_1] \rightharpoonup \cdots \rightharpoonup A_r[j_r]$ on $\bigcup_{i \in k..l} A_{|Q_i}$ is a *marked path in* δ *between* V *and* W *through* O if and only if the following conditions hold:

1. $A_1[j_1] = V$;
2. $A_r[j_r] = W$;
3. $O = \{\mathrm{op}(A_i) \mid i \in 1..r\}$;
4. $\forall i \in 1..r-1 : A_i[j_i] \rightharpoonup A_{i+1}[j_{i+1}]$ being part of the mark implies that at least one of the following is true:
   (a) $A_i = A_{i+1}$ and A_i is a moded unification and j_i is an input position of A_i and $A_i[j_{i+1}] \in \mathcal{V} \cap out(A_i)$.
   (b) $A_i \neq A_{i+1}$ and $A_{i+1}[j_{i+1}] = A_i[j_i] = X \in \mathcal{V} \cap out(A_i) \cap in(A_{i+1})$.

(c) $A_i \in A_{|Q_f}, A_{i+1} \in A_{|Q_{f+1}}, j_{\{}i+1\} = j_i$ and $A_{i+1} = A_i\theta_f$ (thus $\mathrm{op}(A_i) = \mathrm{op}(A_{i+1})$).

(d) $A_i \in A_{|Q_f}, A_{i+1} \in A_{|Q_{f+1}}$, A_i is a call to a predicate p/m defined by a clause $p(X_1, \ldots, X_m) \leftarrow A^p_1, \ldots, A^p_h$, j_i is an input position of A_i $(1 \leq j_i \leq m)$ and $A_{i+1} = A\theta_f$ for some $A \in (A^p_i)_{i \in 1..h}$ where $X_{j_i} \in \mathit{in}(A)$ such that $A[j_{i+1}] = X_{j_i}$.

The conditions in Point 4 of Definition 22 are to be understood as follows. First, whenever an unification presents a marked (input) value or variable, the output variables can be marked. Second, whenever a variable is marked in an output position, the marking can propagate to other (input) occurrences of the same variable. Thirdly, a marked position can be marked again in the same atom (i.e. the atom concerning the same program point in the next version of the query). Finally, when an input position is marked in a predicate call, the mark can further be established on those positions that are occupied by the corresponding formal argument in the predicate's definition.

Example 8

Let us consider the following simple example where two predicates, $p/3$ and $q/2$ are used. Their definitions are the following:

$$
\begin{array}{lcl}
p(X,Y,Z) & \leftarrow & Y := X, q(Y,Z). \\
q(V,W) & \leftarrow & W \Leftarrow f(V).
\end{array}
$$

For the example, we will investigate the derivation yielded by the query $\leftarrow A \Leftarrow 10, p(A,B,C)$.

$$
\begin{array}{cl}
 & \leftarrow A \Leftarrow 10, p(\mathbf{A}, B, C) \\
\overset{\{A/10\}}{\rightarrow} & \leftarrow p(\mathbf{10}, B, C) \\
\overset{\{X/10,Y/B,Z/C\}}{\rightarrow} & \leftarrow \mathbf{B} := \mathbf{10}, q(\mathbf{B}, C) \\
\overset{\{B/10\}}{\rightarrow} & \leftarrow q(\mathbf{10}, C) \\
\overset{\{V/10,W/C\}}{\rightarrow} & \leftarrow \mathbf{C} \Leftarrow f(\mathbf{10}) \\
\overset{\{C/f(10)\}}{\rightarrow} & \diamond
\end{array}
$$

The mark displayed in bold typing above is obtained as follows. First, we mark A in the initial query since this is its last apparition in the queries. We chose to mark the position where A is an input argument (in the call to $p/3$. Next, we propagate this mark using (c) to the same position of the same program point, in the next query where A is now replaced by 10. We use (d) to propagate the mark to a position where the first argument of $p/3$ is input: in the assignment to Y. We use (a) to propagate this mark to B which is output of the assignment. We propagate the mark using (b) from this position to the position $q(B,C)[1]$ where B appears as input. Similarly as before, we propagate the mark using (c), (d) and (a) to reach variable C. This has established the fact that there is a marked path in the derivation between X (the first argument of $p/3$) and Z (its third argument) through $O = \{p, :=, q, \Leftarrow\}$. A contrived version of this mark can be used to find a marked path between X and Y through $\{p, :=\}$.

Now, we can generalize the idea of Definition 22 and Example 8 to the formal arguments of any predicate, using a generic query pattern as in the following definition.

Definition 23

Given a program P and a predicate p/n defined therein with head $p(X_1, \ldots, X_n)$. Let A be the atom equivalent to the head in question, i.e. $A = p(X_1, \ldots, X_n)$. We say that *an argument $V \in (X_i)_{i \in 1..n}$ builds another argument $W \in out(A) \setminus \{V\}$ using op in P* if and only if there exists a successful derivation $\delta = Q \leftarrow \cdots \leftarrow \diamond$ with Q a query of the form $\leftarrow Y_i^1 \Rightarrow t^1, \ldots, Y_i^l \Rightarrow t^l, A$ such that $(Y_i^j)_{j \in 1..l} = in(A) \wedge t^j \in \mathcal{T}$ is ground, and there exists some marked path in δ between V and W through O such that $op \in O$.

The following result states that all the operations linking two formal arguments in the sense of the previous definition are found by our analysis. In it, we use the notation $PA(P)$ to represent the fixpoint of the analysis of a program P, i.e. a function mapping each predicate constituting P onto its computed interactions set.

Theorem 1
Given a program P and a predicate p/n defined therein with head $p(X_1, \ldots, X_n)$. If there exists a couple of variables $V, W \in (X_i)_{i \in 1..n}$ and an operator $op \in B \cup \mathcal{P}$ at program point k such that V builds W using op in P, then $V \overset{Os}{\rightsquigarrow} W \in PA(P)(p)$ where

- $op \in B \implies (op, k) \in Os$
- $op = p/n \implies (\psi_\perp, k) \in Os$
- $op = q/m \in P \setminus \{p/n\} \implies (\psi(\alpha_1, \ldots, \alpha_m), k) \in Os \wedge \forall i \in 1..m, o \in lfp(R), j \in 1..m : (o, j) \in \alpha_i \implies Y_i \overset{Os'}{\rightsquigarrow} Y_j \in PA(P)(q)$ where $o \in Os$ and $q(Y_1, \ldots, Y_m)$ is the head defining q/m.

Proof
The result follows from our construction of the analysis as well as the fact that the program does not allow indirect recursion. As such, q/m in the theorem is necessarily associated to a computed profile that, itself, is sound in the sense of the theorem. □

Proposition 3
The converse of Theorem 1 does not hold, i.e. given a program P and a predicate p/n such that $V \overset{Os}{\rightsquigarrow} W \in PA(P)(p)$, there might exist some $(op, k) \in Os$ such that V does not build W using the operation at program point k in P.

Proof
Let us consider the following predicate defined in some program P.

$$\begin{array}{lll} q(X,Y) & \leftarrow & X \Rightarrow nil, E_1 \Leftarrow 5, E_2 \Leftarrow nil, E \Leftarrow cons(E_1, E_2), E \leftrightarrow X, Y := X. \\ q(X,Y) & \leftarrow & X_1 \Leftarrow 3, X_2 \Leftarrow nil, X \Rightarrow cons(X_1, X_2), Y \Leftarrow nil. \end{array}$$

Our analysis finds the interaction $\{X \overset{\{:=\}}{\rightsquigarrow} Y\}$ in the first clause, but no successful derivation exists for a query $\leftarrow q(X, Y)$ that effectively uses this clause, due to the presence of the ever-failing test $E \leftrightarrow X$ in its body. □

The two above results essentially incarnate the fact that out analysis over-approximates the sets of operations that can relate an input and an output argument. There can therefore be false positives, i.e. cases when the analysis states that some variables interact with each other through an operation that will not happen in actual executions.

D. Time complexity

Theorem 2
Let P be a program containing ℓ_P predicates, with a total of ℓ_a program points. Let $\ell_{io} = \max\{(j+(l-1))\times l \mid p/n \in P, p/n \text{ has } j \text{ input arguments and } l \text{ output arguments}\}$. Then the running time of the analysis is of worst-case complexity $\mathcal{O}(\ell_P \times \ell_{io} \times \ell_a \times \ell_R)$ with ℓ_R a finite natural proportional to the number of potential operations to be registered in the predicates.

Proof
Let us consider the analysis of a given predicate $p_k/n (k \in 1..\ell_P)$. The required lattice for the abstract value associated to the predicate has $\bot$, i.e. $\{\}$, as minimal set of interactions. The maximal element, $\top_p$, is the set containing an interaction $V_i \overset{\mathbb{O}_k}{\rightsquigarrow} V_o$ for each pair of variables $V_i, V_o \in \mathit{args}(p_k)$ such that $i \neq o$ and V_o is output. The elements in-between in the lattice are the sets of "incomplete" interactions, i.e. where all variables and/or operations are not present.

The number of combination of arguments in potential interactions of p_k is $(j+(l-1))\times l$, with j, resp. l, the number of input, resp. output arguments of p_k, since each input argument can have exactly one interaction with each output argument, and each output argument can also contribute to the construction of the $(l-1)$ other output arguments. This quantity is majored by $n - 1 \times n$.

We still need to prove that a finite number of (also finite) operations from $\mathit{lfp}(R)$ suffices to populate the potential interactions and thereby restrict the lattice's height. First, observe that the number of operations in an interaction is majored by the number of program point in P which is finite. Now concerning the ψ-based operations, only a finite amount of these is treated by the analysis as stated earlier. We will denote by ℓ_R the number of operations that the analysis could possibly compute for a predicate given a program's call graph. For p_k, this quantity is proportional to both the number of program points in its body and, recursively, the number of potential operations of the predicates it makes calls to. These ψ-based operations evolve as they are recomputed by successive analysis rounds; ℓ_R represents the number of such steps that can occur before a computed ψ-operation converges. The convergence itself is, of course, guaranteed, since the program call graph cannot contain cycles, so that each predicate's profile is eventually obtained.

So the height of the lattice, that is the maximal number of steps from $\bot$ to $\top_p$, is majored by $\ell_R \times (n-1) \times n$ (this corresponds to adding, at each step up the lattice, an operation to one of the existing interactions, or creating an interaction decorated by one operation). Given that the analysis, at each round, climbs up in the lattice until reaching a fixpoint, this gives a realistic upper bound for the number of analysis iterations for p_k.

The analysis might have to run up the lattice of each of the ℓ_P predicates in P, and at each iteration it needs to crawl through ℓ_a program points and compute ℓ_P projections, hence the result. □